Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 16-003

Trading Timeliness and Accuracy in Geo-Distributed Streaming Analytics

Benjamin Heintz, Abhishek Chandra, Ramesh K. Sitaraman

March 3, 2016

Trading Timeliness and Accuracy in Geo-Distributed Streaming Analytics

Benjamin Heintz
University of Minnesota
Minneapolis, MN
heintz@cs.umn.edu

Abhishek Chandra
University of Minnesota
Minneapolis, MN
chandra@cs.umn.edu

Ramesh K. Sitaraman
UMass, Amherst & Akamai
Tech. Amherst, MA
ramesh@cs.umass.edu

ABSTRACT

Many applications must ingest rapid streams of data and produce analytics results in near-real-time. Whether the input streams represent sensor data from smart homes, user interaction logs from streaming video clients, or server logs from a content delivery network (CDN), it is common for such streams to originate from geographically distributed sources. The typical infrastructure for processing these geo-distributed streams follows a hub-and-spoke model, where several edge resources 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 near-real-time. When this is the case, applications must either sacrifice timeliness by allowing delayed—and in turn *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 the key metrics of 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 in the face of bandwidth limitations. Using a workload derived from a web analytics service offered by a large commercial CDN, we demonstrate the effectiveness of our techniques through a trace-driven simulation. Our results show that our proposed algorithms outperform several baseline algorithms for a range of error and staleness bounds, for a variety of aggregation functions under different network bandwidth constraints.

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, and several scalable systems have been proposed recently as a result [5, 2, 6, 12, 17, 21].

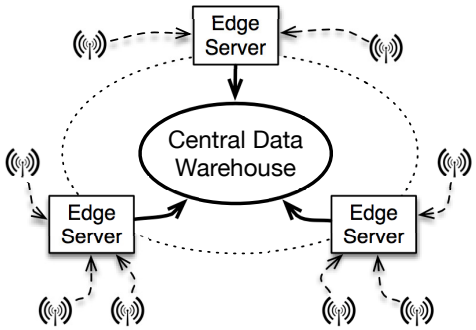


Figure 1: The distributed model for a typical analytics service comprises a single center and multiple edges, connected by a wide-area network in a hub-and-spoke architecture.

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 devices that are physically fixed at diverse locations all around the globe; such data are truly "born distributed" [19]. The distributed infrastructure of a typical geo-distributed analytics service such as Google Analytics or Akamai Media Analytics follows a hub-and-spoke model (see Figure 1). In this architecture, numerous distributed data sources send their streams of data to nearby "edge" servers, which perform partial computation before forwarding results onto 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. There has been recent effort [16, 18, 19] in building geo-distributed analytics systems, both for batch as well as stream computing.

A crucial question for a geo-distributed analytics system is

how to effectively utilize resources both at the edges and at 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, 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 (e.g., Spark Streaming [21]) and batch settings. For example, the 'Group By' construct in SQL allows straightforward expression of grouped aggregation, while grouped aggregation represents the fundamental abstraction underlying MapReduce. Although this pattern may seem restrictive, it is general enough to cover a broad range of applications [6]. For example, windowed grouped aggregation allows a CDN operator to compute average load by region every five minutes in order to quickly identify performance anomalies. A web analytics user interested in identifying trends in content popularity can use windowed grouped aggregation to compute the number of unique clients visiting each URL every hour.

Our prior work [10] examined these questions for 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 [3, 9] 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 [18]. 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 notified of 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), and might 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 the 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, and explore how this tradeoff varies with bandwidth constraints.

• To accommodate diverse application requirements, 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 in practice. 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 [10] by introducing novel cache *partitioning* policies to identify what updates must be sent and which ones can be discarded. This cache abstraction allows us to employ simple and practical design decisions in the choice of cache eviction, cache sizing, and error prediction policies.

• We present a detailed exploration of our design choices as well as insights into the properties of our algorithms through a trace-driven simulation driven by workloads derived from traces of a popular web analytics service offered by Akamai [14], a large content delivery network.

Our simulation results show that our online algorithms significantly outperform baselines of streaming, batching, and batching with random early updates, while approaching the offline optimal algorithms in its achieved error (resp., staleness) under staleness (resp., error) constraints. We find these results to hold for different choices of aggregation functions as well as network bandwidth constraints.

2. PROBLEM FORMULATION

System Model. We consider the typical hub-and-spoke architecture of an analytics system with a center and multiple edges(see Figure 1). Data streams are first sent from each source to a nearby edge. The edges collect and (optionally, partially) aggregate the data. The 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 a promising future direction, but is outside the 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* of the record. 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 that have 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

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

k, an aggregate value $V_k = v_1 \oplus v_2 \oplus \cdots \oplus v_n$ is computed, where $\oplus$ is a user-defined associative binary operator.

To compute windowed grouped aggregation, the distributed infrastructure can perform aggregation at the edge as well as the center. The data 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 $\oplus$ 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 for 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 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 then 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 a key k with $n_k > 0$ arrivals, no update is flushed; i.e., $m_k = 0$. The second is when at least one record arrives for key k after the final update is flushed; i.e., when $d_{m_k,k} < a_{n_k,k}$.

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 $T+W+s$, as illustrated in Figure 2. In other words, staleness quantifies the time elapsed from when the time window completes to when the last update for that time window reaches the center and is included in the final aggregate. Roughly, staleness corresponds to the delay measured from when all the data has arrived to when an analytics user can first receive the results of her grouped aggregation query.

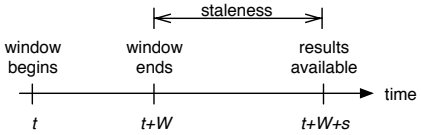


Figure 2: Staleness s is defined as the delay between the end of the window and final results becoming available at the center.

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: $e_k = \text{error}(\hat{V}_k, V_k)$. Note that the particular notion of difference (or error) is application-dependent, and our work applies to both absolute and relative errors. *Error* is then defined as the *maximum* error over all keys: $Error = \max_k e_k$. This 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.

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

3. 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 that staleness for a given time window is defined as the delay between the end of that window and the first time at which all updates for that window have reached the center. Of course 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. Error is caused by any updates that are not delivered to the center, either because the edge sent only partial aggregates or omitted some aggregates altogether.

Intuitively, the two main causes of high staleness are either using too much network bandwidth (causing network congestion and hence, delays due to losses and network queuing), or delaying transmissions till 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* (to avoid transmission delays). 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 for that key, leading to a *residual error* for that key. Thus, we see that there is a fundamental tradeoff between achieving low staleness and low error.

Challenge in optimizing along the tradeoff curve. To understand the challenge of optimizing for either of these metrics while bounding the other, consider two alternate approaches to grouped aggregation: *streaming* that immediately sends all data to the center (without any aggregation at the edge), and *batching* that aggregates all data during a time window on the edge, and only sends out results to the center at the end of the window. When bandwidth is sufficiently high, streaming can deliver extremely low staleness and high accuracy, as arrivals are flushed to the center without additional delay, and all the data reaches the center quickly. When bandwidth is constrained, however, it can lead to *both* 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. It can also lead to high error because it does not distinguish between updates, with less important updates potentially being sent before high value updates. Batching, on the other hand, pro-

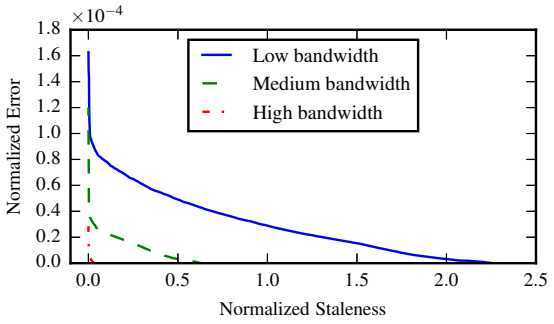


Figure 3: The staleness-error tradeoff curve for a `sum` aggregation over a single time window. Note that in this figure and throughout the paper, we plot staleness normalized relative to the window length and error relative to the maximum possible error for the given data and aggregation function.

vides low error (since it has the final values in hand and can prioritize them based on their importance). It also reduces the network bandwidth consumption, however, it introduces delays by deferring all updates till the end of the window, thus leading to high staleness.

An alternate approach could be to employ a *random sampling* algorithm that sends a subset of aggregate values selected randomly during the window. It would improve over batching by sending updates earlier during the window, thus reducing staleness, and would also improve over streaming under bandwidth constraints by reducing the network traffic. However, it may still lead to high error due to lack of prioritization among different updates. As we will show in Sections 4 and 5, a more principled approach is needed to design an aggregation algorithm that would satisfy the optimization goals specified above.

3.1 Quantifying the Tradeoff

To gain a deeper sense of the staleness-error tradeoff, we use trace-based simulation to compute the complete tradeoff curve over several WAN bandwidths.

Dataset and Simulation Methodology. Throughout this paper, we will use an anonymized workload trace obtained from a real-world analytics service[2] by Akamai, a large commercial content delivery network. This analytics service is used by content providers to track important metrics about who is downloading their content, where these clients are 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"[3]. Each download results in one or more beacons being sent to an edge server, and these beacons contain anonymized 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. Throughout this paper, 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.*

We will focus throughout this paper 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 [10] also considers "smaller" queries; we focus here on the largest queries as they are especially challenging under bandwidth constraints. In this section we consider a `sum` aggregation, which computes the total number of bytes successfully downloaded, and we define error in absolute terms.

To explore the staleness-error tradeoff in detail, we implement a simple discrete event simulator in Python. 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 single-server queuing system 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. For confidentiality reasons, the staleness values are normalized by the window length; the error values reported are absolute errors, and are normalized by the maximum error observed over any window used in the simulation.

Visualizing a Single Window. We use the following approach to explore the complete tradeoff curve. We begin with a staleness-optimal *exact* schedule: an update is flushed for each key immediately upon the last arrival to that key. (For more details on why this is optimal, See our previous work on exact windowed grouped aggregation [10].) We then identify a key—or possibly a set of keys—for which omitting the final arrival yields the smallest increase in error. We omit the final arrival for such key(s), and compute the error and optimal staleness for this new arrival sequence. We iterate this process until reaching zero staleness.

Figure 3 shows this tradeoff curve for a single time window over several WAN bandwidths. These bandwidths correspond to three distinct and meaningful regimes. Here *Low* bandwidth is lower than that required for even optimal traffic under exact computation, while *High* is high enough to allow exact computation without requiring any aggregation at all. *Medium* bandwidth represents bandwidth high enough to support exact computation only when aggregation is performed. Note that the tradeoff curve we see here is the *optimal* tradeoff curve: for any error, it shows us the minimum possible staleness, and for any staleness, it shows us the minimum possible error.

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

[3] 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.

As we would intuitively expect, lower WAN bandwidth leads to higher staleness for a given error, and vice versa. Where bandwidth is sufficiently high, staleness is low even for exact computation (zero error), and only a relatively small tolerance for error is required in order to bring this staleness down to zero. On the other hand, when bandwidth is low, the network is so constrained that exact computation is no longer feasible in near-real-time, and staleness can eventually become unbounded. In this regime, approximation is the only way to maintain near-real-time computation, and Figure 3 shows how our tolerance for error must grow as we demand lower staleness.

Extending to Consecutive Windows. Within the context of a single window, the optimal staleness-error tradeoff curve is unique. When we consider multiple consecutive windows, however, matters are more complicated for two reasons. First, staleness from one window has an effect on the next window. For example, if window n has a staleness of s seconds, then during the first s seconds of window $n+1$, the network is effectively unavailable for transmitting new updates[4]. Second, staleness and error are data-dependent, and the arriving records vary from window to window. For example, the optimal error for a given staleness will tend to be higher for windows with more arrivals, or with arrivals that carry "larger" values with respect to their impact on error, or if the arrivals occur in a burst toward the end of the window. Overall, the relationship between staleness and error for a given window therefore depends both on the data that arrives during that window, and also on the scheduling decisions made during the prior window. Due to these window-to-window differences in the tradeoff between staleness and error, we see different behavior when we apply a staleness bound and minimize error for each consecutive window than when we apply an error bound and minimize staleness for each window.

To demonstrate this behavior in detail, Figure 4 shows staleness for each of 25 consecutive windows in our trace for low, medium, and high WAN bandwidths. Figure 5 shows error for these same 25 windows under the same set of bandwidth constraints. Staleness and error for these figures are based on the offline optimal algorithms we describe in detail in the following section, and they are computed using our discrete event simulator over our anonymized Akamai download analytics trace.

These figures lead to several important observations. First, when bandwidth is low, exact computation is infeasible; applications must tolerate some error. Specifically, Figure 4(a) shows that excessively stringent error constraints lead to unbounded staleness, as staleness accumulates from one window to the next and network delays grow without bound. In other words, when the error constraint is too low, stable near-real-time computation is *infeasible*.

Figure 5(a) reveals how applying a staleness bound as opposed to an error bound leads to different behavior. While we again observe that exact results are infeasible when bandwidth is this low (each of the 25 consecutive windows yields nonzero error) we can see that error from one window has no effect on the next. As a result, it is not necessary to select a feasible staleness bound a priori. Instead, any staleness bound is feasible in the sense that we can always increase error in order to reduce staleness to an arbitrary value.

When bandwidth is constrained, but not extremely so, approximation is not strictly required, but the staleness-error tradeoff can be a very powerful tool for meeting diverse application requirements. For example, Figure 4(b) shows that an error-tolerant application can significantly reduce the variability in staleness. In fact, if an application is sufficiently tolerant of error, it can nearly eliminate staleness. Similarly, Figure 5(b) shows that an application can achieve reliably bounded staleness by tolerating some (possibly intermittent) approximation.

Finally, it is clear that as bandwidth increases, the magnitude of the staleness-error tradeoff decreases. In particular, Figure 4(c) demonstrates that, at high bandwidth, not only is exact computation feasible, but it can be achieved with relatively low staleness. Figure 5(c) shows that even a small tolerance for staleness is sufficient to allow exact computation. In other words, when bandwidth is very high, there is relatively little room for exploiting the staleness-error tradeoff.

4. OFFLINE ALGORITHMS

We now consider the two complementary optimization problems of minimizing staleness (resp., error) under error (resp., staleness) constraint. Before we present practical online algorithms to solve these problems, we consider optimal *offline* algorithms for these problems. These offline algorithms 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.

4.1 Constrained Error: Minimizing Staleness

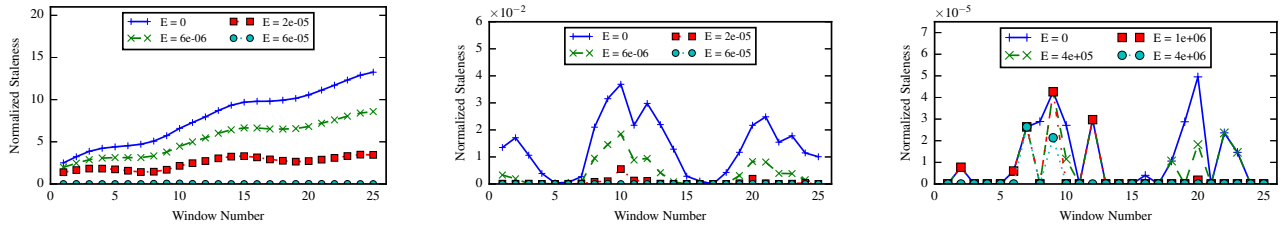
The first optimization problem we consider is minimizing staleness under an error constraint (s.t. error $\leq E$), where E is an application-specified error tolerance value.

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* $V_i(t)$ for key i at time t to be the aggregate value of all arrivals for key i during the current window prior to time t. We define the prefix aggregate $V_i(t)$ to have a logical zero value prior to the first arrival for key i. 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* $e_i(t)$ for key i at time t to be the error of the prefix aggregate for key i with respect to the true final aggregate: $e_i(t) = \text{error}(V_i(t), V_i(T+W))$. We refer to the prefix error of key i at the beginning of the window—$e_i(T)$—as the *initial prefix error* of key i.

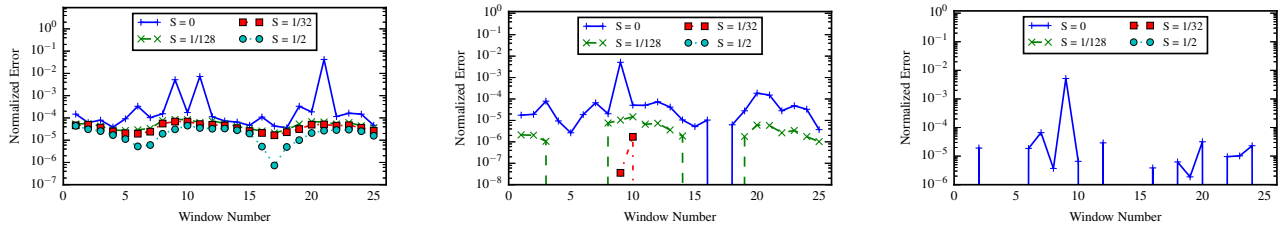
THEOREM 1 (EAGER PREFIX ERROR). *Given an error constraint E, an optimal algorithm flushes each key i at the first time t such that $e_i(t) \leq E$. If $e_i(T) \leq E$, then an optimal algorithm avoids flushing key i altogether.*

[4]We assume a FIFO ordering of data records over the network, as is typically the case with protocols like TCP.



(a) Low bandwidth. (b) Medium bandwidth. (c) High bandwidth.

Figure 4: Staleness for 25 consecutive windows with various error constraints.



(a) Low bandwidth. (b) Medium bandwidth. (c) High bandwidth.

Figure 5: Error for 25 consecutive windows with various staleness constraints. Note the logarithmic y-scale.

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. □

We refer to this algorithms as the *Eager Prefix Error (EPE)* algorithm.

COROLLARY 2. *When $E = 0$, the EPE algorithm achieves optimal staleness.*

This is because, when $E = 0$, this algorithm flushes an update for each key upon the final arrival for that key[5]. It is therefore strictly a generalization of the eager offline optimal algorithm for *exact* windowed grouped aggregation [10], which is staleness-optimal.

COROLLARY 3. *Because this algorithm flushes only keys with $e_i(T) > E$, it is also traffic optimal for any given error bound E.*

4.2 Constrained Staleness: Minimizing Error

Next, we consider the optimization problem of minimizing error under a staleness constraint (s.t. 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[6]. If S denotes the staleness constraint in seconds, then the final slot ends S seconds after the end of the window. 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 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.

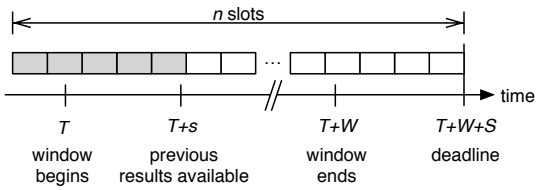


Figure 6: We view the network as a sequence of contiguous slots. Shaded slots are unavailable to the current window due to the previous window's staleness.

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. Figure 6 illustrates our model for the time window $[T, T + W)$.

To understand how we assign keys to slots, we must first introduce the notion of *potential error*. The potential error $E_i(t)$ for key i at time t is defined to be the error that the center would see for key i if key i were assigned to a slot beginning at time t. Recall that, for $t < T + s$, slots

[5]The flush can occur prior to the final arrival if the arrivals for key k end with a sequence of one or more zeroes.
[6]In general, bandwidth need not be constant.

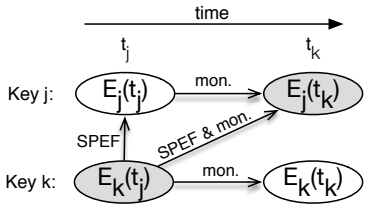


Figure 7: Smallest-potential-error-first ordering minimizes error.

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, and making such an assignment is therefore 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

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

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

An optimal 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.

LEMMA 4. *Assigning keys in smallest-potential-error-first (SPEF) order minimizes error.*

PROOF. For each key $1 \leq i \leq n$, let t_i be the (only) time at which key i is flushed. The final error is then given by $E = \max_{1 \leq i \leq n} E_i(t_i)$. To begin, assume that potential error $E_i(t)$ is monotonically decreasing in t.

Consider a schedule that flushes keys j and k at times t_j and t_k respectively, where $t_j < t_k$. Further assume that these keys are scheduled in smallest-potential-error-first order; i.e., that $E_j(t_j) \leq E_k(t_j)$. The final error E is then bounded by $E \geq \max(E_j(t_j), E_k(t_k))$.

Alternatively, consider a schedule that swaps these keys, flushing key j at time t_k, and key k and time t_j. This would yield error $E' \geq \max(E_k(t_j), E_j(t_k))$.

Figure 7 illustrates the relationships between these different errors. The non-shaded vertices contribute to E, while the shaded vertices contribute to the alternative error E'. To see why $E' \geq E$, first consider E'. By monotonicity and SPEF ordering, we know that $E_k(t_j) \geq E_j(t_k)$, so it is key k that contributes the greater error to E', and we can describe E' more simply by $E' \geq E_k(t_j)$.

This error cannot be lower than E because, by SPEF ordering, $E_k(t_j)$ is no smaller than $E_j(t_j)$, and by monotonicity, $E_k(t_j)$ is no smaller than $E_k(t_k)$. In other words, whether key j or key k contributes more to the final error in the SPEF-ordered schedule, $E' \geq E$. □

LEMMA 5. *An optimal schedule D_{opt} that flushes only $m < n$ keys can be transformed into another optimal schedule D'_{opt} that flushes n keys.*

PROOF. D_{opt} omits $n-m$ keys. Inserting these $n-m$ keys into the slots left vacant by D_{opt} yields D'_{opt}. This transformation cannot increase error, because, by monotonicity, sending a key never yields higher error than omitting it does. Further, it cannot increase staleness due to our slot construction. Hence, D'_{opt} is also optimal. □

THEOREM 6 (SMALLEST POTENTIAL ERROR FIRST). *There exists an optimal algorithm that assigns to each consecutive slot a key with the smallest potential error among all unassigned keys.*

PROOF. Such an 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 D_{opt} and D_{SPEF} are identical until index m, where they first differ. 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 5.) 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 4, we can transform D_{opt} into D'_{opt} by swapping keys m and p without increasing error. 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}. □

We refer to the above algorithm as the *Smallest Potential Error First (SPEF)* algorithm.

COROLLARY 7. *Let S_{opt} denote the optimal staleness for exact computation. When the staleness constraint $S \geq S_{\text{opt}}$, the SPEF algorithm achieves zero error.*

Intuitively, this is because for an optimal staleness bound S_{opt} (or higher), by definition, the first of the n slots for key updates begins late enough that there always exists a key that has attained its final value. The SPEF algorithm then always selects one of these exact keys to be flushed, and hence achieves zero error. It, in fact, reduces to the lazy optimal algorithm [10] for *exact* computation for an optimal staleness bound.

Up to this point, we have assumed that prefix error decreases monotonically in time, but this assumption can be relaxed in a straightforward manner. Consider an aggregation for which prefix error is non-monotonic. Let the *minimum potential error* for key i at time t be the smallest potential error achieved up until time t during the current window. This value decreases monotonically whether or not the underlying error is monotonic. Therefore the following must hold.

COROLLARY 8. *If flushed updates always contain the value that achieves this minimum potential error, then assigning to each slot the key with the smallest* minimum potential error *is an optimal algorithm.*

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.

LEMMA 9. *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.*

PROOF. A key is omitted when it is assigned to a slot prior to the first arrival for that key, or before the network is available. In either case, its potential error is equal to its initial prefix error. SPEF assigns keys in smaller-potential-error-first (SPEF), so at the time that key i is omitted, its initial prefix error must be smaller than the potential error of all other not-yet-assigned keys. By monotonicity, potential error never exceeds initial prefix error, so key i must also have smaller initial prefix error than all other not-yet-assigned keys. □

It is important to note that this is a necessary but insufficient condition for a key to be omitted: another key may have a smaller potential error than the smallest initial prefix error.

We can now apply this property to derive an alternative optimal algorithm, referred to as *SPEF with early omissions (SPEF-EO)*. Let m be the number of keys that our original optimal algorithm omits.

THEOREM 10 (SPEF WITH EARLY OMISSIONS). *There exists an optimal algorithm that first omits the m keys with the smallest initial prefix errors, then assigns the remaining $n - m$ keys in SPEF order.*

PROOF. By Lemma 9, 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. □

4.3 High-Level Lessons

These offline optimal algorithms, although not applicable in practical online settings, leave us with several high-level lessons. First, these optimal algorithms (EPE as well as SPEF/SPEF-EO) send at most one flush per key, thereby avoiding wasting scarce network resources. Second, each algorithm makes 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.

5. ONLINE ALGORITHMS

We are now prepared to consider practical online algorithms for achieving near-optimal staleness-error tradeoffs. The offline optimal algorithms from Section 4 serve as useful baselines for comparison, and they also provide models to emulate. Throughout this section, we will explore different design choices in designing our practical algorithms using trace-driven simulations with a real-world dataset. All simulation results shown in this section present 95th percentile values (staleness or error) over 25 consecutive time windows from the Akamai trace, spanning multiple days worth of workload.

5.1 The Two-Level Cache Abstraction

Exact Computation. Our online algorithms for approximate windowed grouped aggregation generalize those for exact computation [10], 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 cache until they have reached their final value, while avoiding holding them so long as to lead to high staleness. The eviction policy, meanwhile, determines *what* 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 a *primary cache* and a *secondary cache*. 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 at all. 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 choice of partitioning policy. As we will 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 serves as the outgoing network queue, and updates are pulled 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 point in time.

5.2 Error-bound Algorithms

As discussed above, the two key design choices are when to promote an aggregate from the secondary to the primary queue, and what eviction policy to use to flush updates from the primary cache to the network.

Key Promotion. 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 4.1) 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_i denote the total aggregate value flushed for key i so far during the current window (logically zero if no update has been flushed), and let V_i denote the aggregate value currently maintained in the secondary cache for key i. Then the *accumulated error* for key i is defined to be the error between the value that the center currently knows (F_i) and the value it would see if key i were flushed ($F_i \oplus V_i$). Key i is moved from the secondary cache to primary cache when error$(F_i, F_i \oplus V_i) > E$. Given this policy, and the fact that updates are only flushed from the primary cache, we are guaranteed to flush only keys that *must* be flushed in order to satisfy the error constraint.

Intuitively, this approach guarantees that the primary cache only contains keys that we are certain must be flushed; it avoids false positives. On the other hand, this approach only *eventually* avoids false negatives. That is, a key that will eventually need to be flushed may remain in the secondary cache beyond the time at which its prefix error falls below the error bound E. A more *eager* variant would promote a key into the primary cache when it is probable—but not necessarily certain—that it will require flushing. We do not explore such alternatives in depth here, however, as our simulation results show that these transient false negatives contribute only a small amount to staleness.

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[7].

Cache Eviction Policy. Prior to the end of the window, some prediction is involved. In particular, when the network pulls an update from the primary cache, it is the task of the eviction policy to identify a key that has reached an aggregate value within E of its final value. We explore three practical cache replacement policies: least-recently used (LRU), smallest potential error (SPE), and most accumulated error (MAE). SPE tries to faithfully emulate the offline optimal algorithm by explicitly predicting the final value for each key, and uses this to estimate the potential error of each key, ultimately evicting the key with the smallest estimated potential error. The MAE policy is greedy in nature: it evicts the key that will make the largest reduction in error; i.e., the key with the largest accumulated error currently.

Figure 8 compares these simple and practical eviction policies. We find that LRU and SPE perform about equally and both outperform MAE, especially at lower error limits. LRU

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

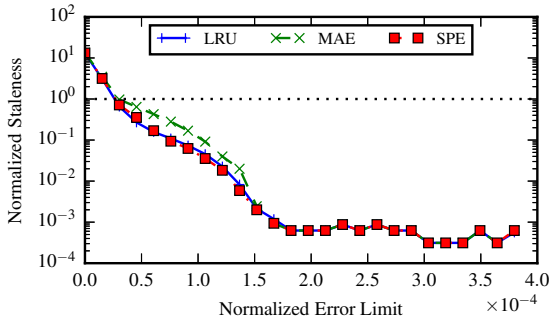


Figure 8: Impact of primary cache eviction policy on staleness given an error bound. Note the logarithmic y-scale.

may be preferable to SPE because of its simplicity. The main point of these results is that it is not necessary to develop a sophisticated prediction policy to yield low staleness under an error constraint. Instead, even simple and practical policies are very effective.

5.3 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 4.2). To understand this approach, recall Theorem 10, which shows that an optimal algorithm flushes only the keys with the largest initial prefix errors. We emulate this behavior by logically ranking cached keys by their (estimated) initial prefix error, and defining the primary cache at time t to comprise the top $s(t)$ keys in this order. In practice, this is challenging as we must predict both initial prefix errors as well as an appropriate sizing function $s(t)$. We first discuss the choice of primary cache eviction policy followed by effective practical approaches to these challenges.

Cache Eviction Policy. During the window, we use this 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 5.2, we again find that well known and relatively simple eviction policies such as LRU perform very well.

Upon reaching the end of the window, there is no longer any 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.

Initial Prefix Error Prediction. During the window, there are many ways to predict initial prefix errors. One straightforward approach is to use accumulated error as a proxy for initial prefix error. We call this the *Acc* policy for short.

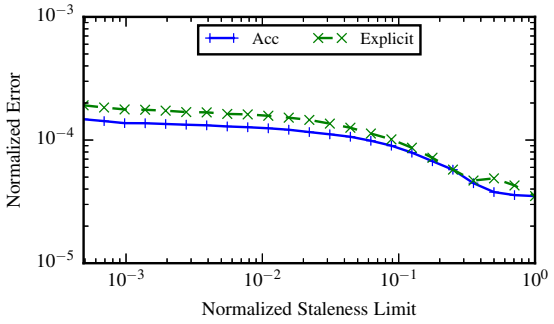


Figure 9: Impact of the initial prefix error prediction algorithm on staleness given error bound. Note the logarithmic axis scales.

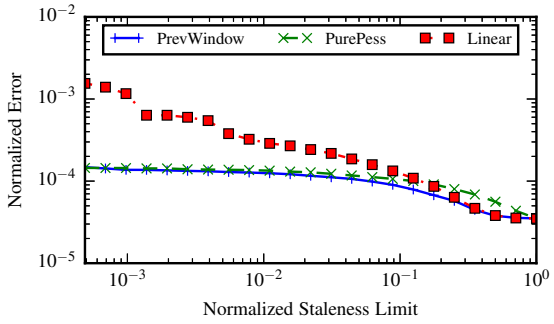


Figure 10: The impact of primary cache sizing policy on error given staleness bound. Note the logarithmic axis scales.

An alternative approach—which we simply call *Explicit*—attempts to explicitly predict initial prefix errors by predicting the final value for each key.

Figure 9 compares these approaches, and shows that the very simple Acc policy yields lower error at each staleness bound than a more complex approach does. It is also more attractive since it only relies on values seen so far. This again demonstrates that sophisticated prediction approaches are not required to yield good performance with our two-part caching approach; even very simple and practical policies can be effective.

Cache Size Prediction. Predicting the appropriate primary cache size function $s(t)$ 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 $s(t)$ represents the number of keys that are *present* in the primary cache. In other words, it does not include keys with large initial prefix errors that should eventually occupy the primary cache but that have not yet arrived. 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 $B(t)$, the ideal primary cache size function must satisfy $B(t) = s(t) + f(t)$.

In practice, we have neither a perfect eviction policy, nor a perfect prediction of initial prefix error. Further, determining $f(t)$ is a nontrivial prediction challenge on its own. We have explored a host of alternative approaches and we highlight three here: *PrevWindow*, *Linear*, and *PurePess*. The PrevWindow policy assumes that arrivals in the current window will be identical to those in the previous window, and it therefore uses the previous window's arrival history to predict $f(t)$. Additionally, this policy maintains a count of the maximum number of flushes for any single key during the current window, and makes the conservative assumption that all keys in the primary cache will be flushed this many times. The Linear policy makes the simplifying assumption that keys with the largest initial prefix errors have initial arrivals distributed uniformly throughout the window. In other words, if we assume k such keys will arrive during the window, then for the time window $[T, T + W)$, we assume $f(t)$ decays linearly as $f(t) = \frac{k(T-t)}{W}$. The PurePess policy makes a pessimistic assumption about future arrivals, specifically that all future arrivals are for keys that should occupy the primary cache. We monitor the arrival rate of new keys using an exponentially weighted moving average, and base $f(t)$ on an extrapolation of this average rate. As a result, when bandwidth is highly constrained, the primary cache is effectively nonexistent, and this policy degrades to evicting keys in order of estimated initial prefix error.

Figure 10 compares these policies. The figure shows that the PrevWindow policy nearly universally outperforms the others. PurePess also performs reasonably well, but is slightly worse than PrevWindow. The Linear policy is generally outperformed by both PurePess and PrevWindow.

6. COMPARATIVE EVALUATION

In this section, we compare our online algorithms presented above to a number of baseline algorithms. We also investigate the impact of network bandwidth constraints on the performance of these algorithms, as well as the choice of aggregation function. We use the Akamai trace and the trace-driven simulation methodology described in Section 3. As mentioned there, all simulation results present 95th percentile values (staleness or error) over 25 consecutive time windows from the Akamai trace, spanning multiple days worth of workload.

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:* The optimal algorithms consist of Oracle algorithms that have full knowledge of future arrivals, and are able to accurately emulate the offline algorithms (from Section 4). In particular, we use the trace-driven simulation to faithfully emulate the optimal offline EPE and SPEF-EO algorithms for the error-bound and staleness-bound optimizations respectively.
• *Caching:* These refer to our caching-based online algorithms presented in Section 5. For the error-bound case, we use the emulated EPE algorithm, consisting of the two-part caching approach with an LRU primacy cache eviction policy, as discussed in Section 5.2. For the staleness-bound case, we employ the emulated SPEF-EO algorithm discussed in Section 5.3, consisting of LRU eviction, Accumulated error-based estimation, and PrevWindow cache sizing policies.

Note that we use the relevant caching-based and baseline algorithms for comparison for the error-bound or the staleness-bound optimization problems, as appropriate.

6.1 Constrained Error: Minimizing Staleness

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 11 shows the normalized staleness for a range of error bounds E for our online algorithm ('Caching' in the figure), as well as for our three baseline algorithms and an optimal offline algorithm. It shows the comparison for three different aggregation functions under low bandwidth: `sum`, `max`, and `count`. For all aggregations, 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 over batching, but the improvement is minor for `sum` and `max`, which demonstrates the limitation of a 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.

We next examine the impact of bandwidth constraint by running the algorithms under medium and high bandwidth (3 and 10 times higher bandwidth than that used for Figure 11, respectively). Figure 12(a) shows the 95th percentile staleness for a range of error bounds for `sum` aggregation under medium bandwidth. We now see that the magnitude of the error values comes down for all algorithms as compared to the low bandwidth case (Figure 11(a)). However, our caching-based algorithm still outperforms the other baselines and gets close to the optimal algorithm. Figure 12(b) shows that, under high bandwidth, it is possible to simply stream arrivals directly to the center, and each algorithm other than batching yields low staleness.

6.2 Constrained Staleness: Minimizing Error

Figure 13 demonstrates the effectiveness of our approach compared to the other baselines for the staleness-bound case where we are minimizing the error given a staleness limit. The figure shows the normalized error values obtained for different normalized staleness limits, for `sum`, `max`, and `count` aggregations with low bandwidth. 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 flushes represents a further improvement, though again, except for `count` aggregation, the benefit of these early flushes 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 staleness 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 changes to flush updates during the window, and these algorithms converge toward pure batching.

Figure 14 shows the same algorithms applied at medium and high bandwidth for the `sum` aggregation, where the same general observations hold. Echoing the results for our error-bound algorithms in Figure 12, Figure 14 shows that our caching-based algorithm outperforms the other baselines whether or not bandwidth is constrained, with little need for optimization under high bandwidth.

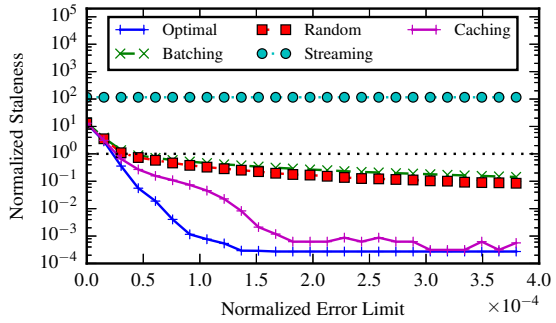
7. RELATED WORK

Numerous streaming systems [5, 7, 12, 17, 21] 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 that we've described here, in particular how to strike a near-optimal balance between timeliness and accuracy.

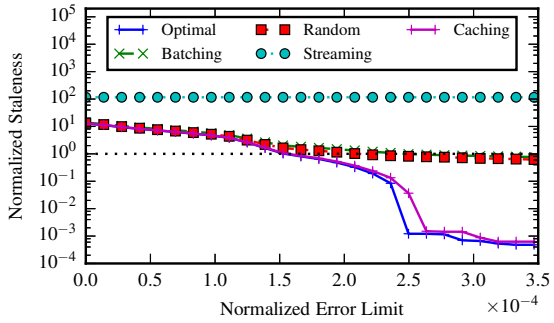
Google recently proposed the Dataflow model [1] as a unified abstraction for computing over both bounded and unbounded datasets. Our work fits within this model, but we focus in particular on aggregation over tumbling windows on unbounded data sets, and our notion of staleness implies a focus on the processing-time domain.

Wide-area computing has received increased research attention in recent years, due in part to the widening gap between data processing and communication costs. Much of this attention has been paid to batch computing [16, 19]. Relatively little work on streaming computation has focused on wide-area deployments, or associated questions such as where to place computation. Pietzuch et al. [15] optimize operator placement in geo-distributed settings to balance between system-level bandwidth usage and latency. Hwang et al. [11] rely on replication across the wide area in order to achieve fault tolerance and reduce straggler effects.

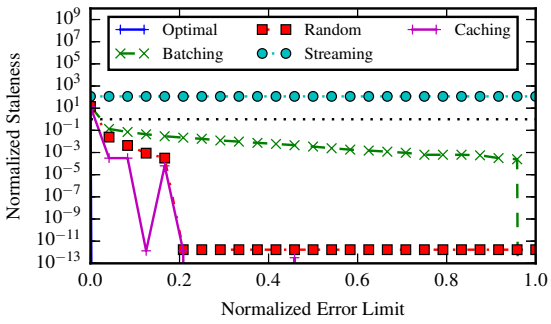
JetStream [18] considers wide-area streaming computation, and like our work, addresses the tension between timeliness and accuracy. Unlike our work, however, it focuses at a higher level on the appropriate abstractions for navigating this tradeoff. Meanwhile BlinkDB [4] provides mechanisms



(a) `sum` aggregation.

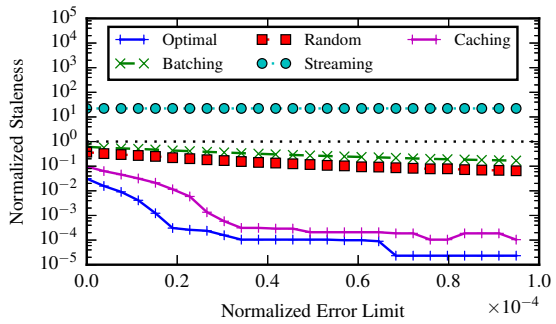


(b) `max` aggregation.

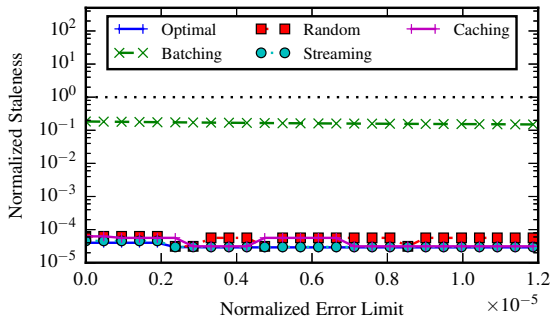


(c) `count` aggregation.

Figure 11: 95th percentile staleness for various error bounds under low bandwidth. Note the logarithmic y-scale.

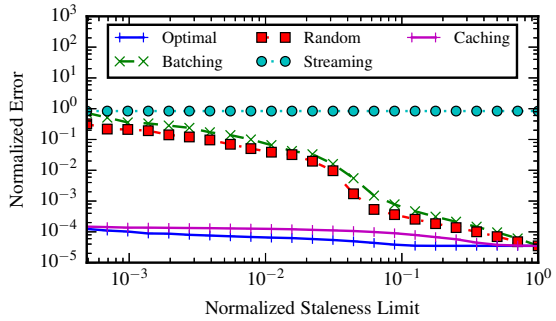


(a) Medium bandwidth.

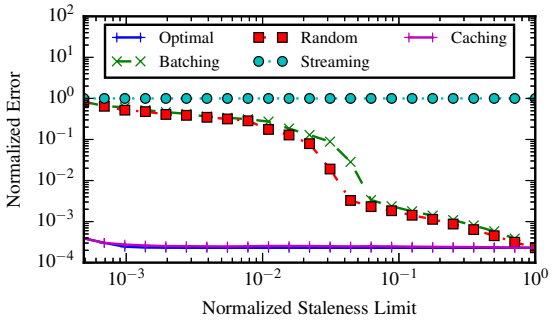


(b) High bandwidth.

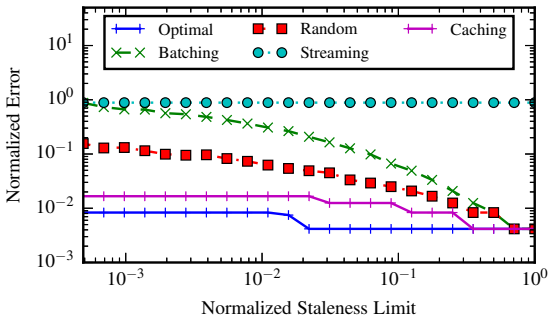
Figure 12: 95th percentile staleness for various error bounds under medium and high bandwidth. Note the logarithmic y-scale.



(a) `sum` aggregation.

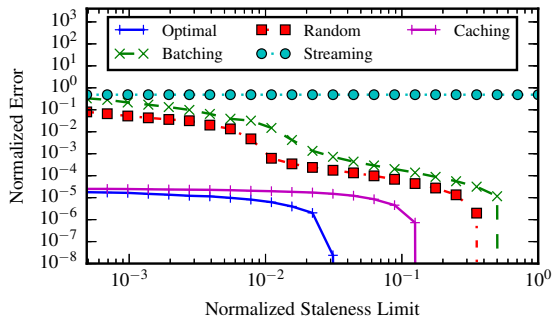


(b) `max` aggregation.

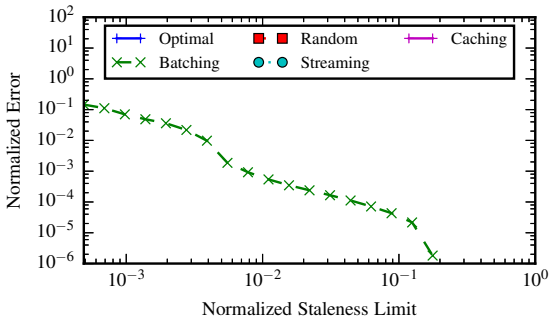


(c) `count` aggregation.

Figure 13: 95th percentile error for various staleness bounds under low bandwidth. Note the logarithmic axis scales.



(a) Medium bandwidth.



(b) High bandwidth.

Figure 14: 95th percentile error for various staleness bounds under medium and high bandwidth. Note the logarithmic axis scales.

to trade accuracy and response time, though it does not focus on processing streaming data. Our focus is on specific policies to approach an optimal tradeoff between timeliness and accuracy.

Aggregation is a key operator in analytics, and grouped aggregation is supported by many data-parallel programming models [6, 8, 20]. Larson et al. [13] 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.

8. 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 present 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 a trace-driven simulation. Our results showed that our proposed algorithms outperform several baseline algorithms for a range of error and staleness bounds, for a variety of aggregation functions under different network bandwidth constraints.

9. REFERENCES

[1] 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*, 8:1792–1803, 2015.

[2] Storm, distributed and fault-tolerant realtime computation. `http://storm.apache.org/`, 2015.

[3] 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.

[4] 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.

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

[6] 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.

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

[8] 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.

[9] 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.

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

[11] 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.

[12] S. Kulkarni et al. Twitter heron: Stream processing at scale. In *Proc. of SIGMOD*, SIGMOD '15, pages 239–250, 2015.

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

[14] 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.

[15] P. Pietzuch et al. Network-aware operator placement for stream-processing systems. In *Proc. of ICDE*, 2006.

[16] 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.

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

[18] 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.

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

[20] 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.

[21] 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.