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 15-001

Optimizing Grouped Aggregation in Geo-Distributed Streaming Analytics

Benjamin Heintz, Abhishek Chandra, and Ramesh K. Sitaraman

January 26, 2015

Revised: February 4, 2015

Optimizing Grouped Aggregation 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

Large quantities of data are generated continuously over time and from disparate sources such as users, devices, and sensors located around the globe. This results in the need for efficient geo-distributed streaming analytics to extract timely information. A typical analytics service in these settings uses a simple hub-and-spoke model, comprising a single central data warehouse and multiple edges connected by a wide-area network (WAN). A key decision for a geo-distributed streaming service is how much of the computation should be performed at the edge versus the center. In this paper, we examine this question in the context of *windowed grouped aggregation*, an important and widely used primitive in streaming queries. Our work is focused on designing aggregation algorithms to optimize two key metrics of any geo-distributed streaming analytics service: *WAN traffic* and *staleness* (the delay in getting the result). Toward this end, we present a family of optimal offline algorithms that *jointly minimize both staleness and traffic*. Using this as a foundation, we develop practical online aggregation algorithms based on the observation that grouped aggregation can be modeled as a *caching* problem where the cache size varies over time. This key insight allows us to exploit well known caching techniques in our design of online aggregation algorithms. We demonstrate the practicality of these algorithms through an implementation in Apache Storm, deployed on the PlanetLab testbed. The results of our experiments, driven by workloads derived from anonymized traces of a popular web analytics service offered by a large commercial CDN, show that our online aggregation algorithms perform close to the optimal algorithms for a variety of system configurations, stream arrival rates, and query types.

1. INTRODUCTION

Data analytics is undergoing a revolution: both the volume and velocity of analytics data are increasing at a rapid rate. Across a large number of application domains that include web analytics, social analytics, scientific computing, and energy analytics, large quantities of data are generated continuously over time in the form of posts, tweets, logs, sensor readings, etc. A modern analytics service must provide real-time analysis of these data streams to extract meaningful and timely information for the user. As a result, there has been a growing interest in streaming analytics with recent development of several distributed analytics platforms [2, 8, 26].

In many streaming analytics domains, data is often derived

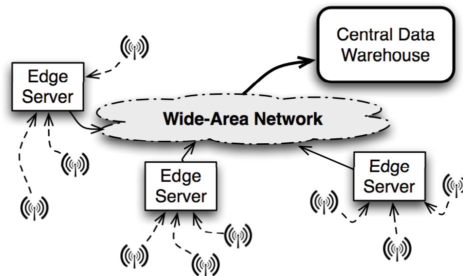


Figure 1: The distributed model for a typical analytics service comprises a single center and multiple edges, connected by a wide-area network.

from disparate sources that include users, devices, and sensors located around the globe. As a result, the distributed infrastructure of a typical analytics service (e.g., Google Analytics, Akamai Media Analytics, etc.) has a hub-and-spoke model (see Figure 1). The data sources generate and send a stream of data to "edge" servers near them. The edge servers are geographically distributed and process the incoming data and send it to a central location that can process the data further, store the summaries, and present those summaries in visual form to the user of the analytics service. While the central location that acts as a hub is often located in a well-provisioned data center, the resources are typically limited at the edge locations. In particular, the available WAN bandwidth between the edge and the center might be limited.

A traditional approach to analytics processing is the *centralized model* where no processing is performed at the edges and all the data is sent to a dedicated centralized location. However, such an approach is often inadequate or suboptimal, since it can strain the scarce WAN bandwidth available between the edge and the center, cause longer delays due to the high volumes of unaggregated data to be sent over the network, and does not make use of the available compute and storage resources at the edge. An alternative is a *decentralized approach* [21] that utilizes the edge for much of the processing in order to minimize the amount of WAN traffic. In this paper, we argue that analytics processing must utilize *both* edge and central resources in a carefully coordinated manner in order to achieve the stringent requirements of an analytics service in terms of both network traffic and user-perceived delay.

An important primitive in any analytics system is *grouped aggregation*. Grouped aggregation is used to combine and summarize large quantities of data from one or more data streams. As a result, it is provided as a key operator in most data analytics frameworks, such as the Reduce operation in MapReduce, or GroupBy in SQL and LINQ. A common variant of the primitive in stream computing is *windowed grouped aggregation* where data produced within finite specified time windows must be summarized. Windowed grouped aggregation is one of the most frequently used primitives in an analytics service and underlies queries that aggregate a metric of interest over a time window. For instance, a web analytics user may wish to compute the total visits to his/her web site broken down by country and aggregated on an hourly basis to gauge the current content popularity. Similarly, a network operator may want to compute the average load in different parts of the network every 5 minutes to identify hotspots. In these cases, the user would define a standing windowed grouped aggregation query that generates results periodically for each time window (every hour, 5 minutes, etc.).

Our work is focused on designing algorithms for performing windowed grouped aggregation in order to optimize the two key metrics of any geo-distributed streaming analytics service: *WAN traffic* and *staleness* (the delay in getting the result for a time window). While much of the existing work on decentralized analytics [21, 23] has focused primarily on optimizing a single metric (e.g., network traffic), it is important to examine both traffic and staleness together to achieve both cost savings as well as higher information quality. The key decision that our algorithms make is *how much of the data aggregation should be performed at the edge versus the center*. To understand the challenge, consider two alternate approaches to grouped aggregation: *pure streaming*, where all data is immediately sent from the edge to the center without any edge processing; and *pure batching*, where all data during a time window is aggregated at the edge, with only the aggregated results being sent to the center at the end of the window. Pure batching results in a greater level of edge aggregation, resulting in a reduction in the edge-to-center WAN traffic compared to pure streaming. However, the edge must wait longer to collect more data for aggregation, risking the possibility of the aggregates reaching the center late, resulting in greater staleness. We have shown [12] that the decision about how much aggregation to be performed at the edge cannot be made statically; rather it depends on several factors such as the query type, network constraints, data arrival rates, etc. Further, these factors vary significantly over time (see Figure 3(b)), requiring the design of algorithms that can adapt to changing factors in a dynamic fashion.

Research Contributions

• To our knowledge, we provide the first algorithms and analysis for optimizing grouped aggregation, a key primitive, in a wide-area streaming analytics service. In particular, we show that simpler approaches such as pure streaming or batching do not jointly optimize traffic and staleness, and are hence suboptimal.

• We present a family of optimal offline algorithms that *jointly minimize both staleness and traffic*. Using this as a foundation, we develop practical online aggregation algorithms that emulate the offline optimal algorithms.

• We observe that grouped aggregation can be modeled as a *caching problem* where the cache size varies over time. This key insight allows us to exploit well-known caching algorithms in our design of online aggregation algorithms.

• We demonstrate the practicality of these algorithms through an implementation in Apache Storm [2], deployed on the PlanetLab [1] testbed. Our experiments are driven by workloads derived from anonymized traces of a popular web analytics service offered by Akamai [17], a large content delivery network. The results of our experiments show that our online aggregation algorithms simultaneously achieve traffic within 2.0% of optimal while reducing staleness by 65% relative to batching. We also show that our algorithms are robust to a variety of system configurations (number of edges), stream arrival rates, and query types.

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 proximal edge. The edges collect and (potentially, partially) aggregate the data. The aggregated data can then be sent from the edges to the center where more aggregation could happen. The final aggregated results are available at the center. Users of the analytics service query the center to visualize the data. 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.

Data Streams and Grouped Aggregation. 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.

Windowed grouped aggregation over a *time window* $[t, t+W)$, where W is the window size, is 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 $\{(k, v_i)\}, 1 \leq i \leq n$, that correspond to the n records in the time window that have key k, an aggregate value $\hat{v} = v_1 \oplus v_2 \cdots \oplus v_n$ is computed, where $\oplus$ is any associative binary operator[1]. Examples of aggregates include sums, histograms, and approximate data types such as Bloom filters. Customarily, the timeline is subdivided into non-overlapping intervals of size W and windowed group aggregation is computed on each such window[2]. Note that the aggregate record is typically of the same size as an incoming

[1]More formally, any binary operator that forms a semigroup can be used.
[2]Such non-overlapping time windows are often called *tumbling* windows in analytics terminology.

data record. Thus, grouped aggregation results in a reduction in the amount of data.

To compute windowed grouped aggregation, we consider aggregation at the edge as well as the center. The data records that arrive at the edge can be partially aggregated locally at the edge, so that 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.

Optimization Metrics. Our goal is to *simultaneously* minimize two metrics: *staleness*, a key measure of information quality; and *network traffic*, a key measure of cost. Staleness is defined as the smallest time s such that the results of grouped aggregation for time window $[t, t+W)$ are available at the center at time $t+W+s$, as illustrated in Figure 2. In our model, staleness is simply 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 is the delay measured from when all the data has arrived to when an analytics user views the results of her grouped aggregation query. The network traffic is measured by the number of updates sent over the network from the edge to the center.

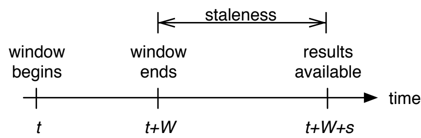


Figure 2: Staleness s is defined as the delay between the end of the window and final results becoming available at the center.

Algorithms for 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 arrivals in the time window, suppose that the ith 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 key k, the output of the aggregation algorithm is a sequence of m_k updates where the ith update $(k, \hat{v}_{i,k})$ departs for the center at time $d_{i,k}$, $1 \leq i \leq m_k$. The updates must have the following properties:

• Each update for each key k aggregates all values for that key in the current time window that have not been previously aggregated.

• Each key k that has $n_k > 0$ arrivals must have $m_k > 0$ updates such that $d_{m_k,k} \geq a_{n_k,k}$. That is, each key with an arrival must have at least one update and the last update must depart after the final arrival so that all the values received for the key have been aggregated.

The goal of the aggregation algorithm is to minimize traffic which is simply the total number of updates, i.e., $\sum_k m_k$. The other simultaneous goal is to minimize staleness which is the time for the final update to reach the center, i.e., the update with the largest value for $d_{m_k,k}$, to reach the center[3].

3. DATASET AND WORKLOAD

To derive a realistic workload for evaluating our aggregation algorithms, we have used anonymized workload traces obtained from a real-life analytics service[4] offered by Akamai which operates a large content delivery network. The download analytics service is used by content providers to track important metrics about who is downloading their content, from where is it being downloaded, what was the performance experienced by the users, how many downloads completed, etc. The data source is a software called download manager that is installed on mobile devices, laptops, and desktops of millions of users around the world. The download manager is 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 edge servers using "beacons"[5]. Each download results in one or more beacons being sent to an edge server containing information pertaining to that download. The 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 we normalize derived values from the data set such as data sizes, traffic sizes, and time durations, for confidentiality reasons.

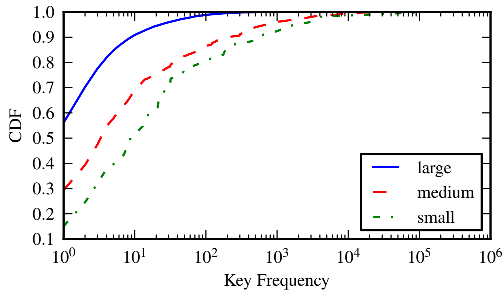
Throughout our evaluation, we compute grouped aggregation for three commonly-used queries in the download analytics service. Queries that are issued in the download analytics service that use the grouped aggregation primitive can be roughly classified according to *query size* that is defined to be the number of distinct keys that are possible for that query. We choose three representative queries for different size categories (see Table 1). The `small` query groups by two dimensions with the key consisting of the tuple of content provider id and the user's last mile bandwidth classified into four buckets. The `medium` query groups by three dimensions with the key consisting of the triple of the content provider id, user's last mile bandwidth, and the user's country code. The `large` query groups by a different set of three dimensions with the key consisting of the triple of the content provider id, the user's country code, and the url accessed. Note that the last dimension—url—can take on hundreds of thousands of distinct values, resulting in a very large query size.

The total arrival rate of data records across all keys for all

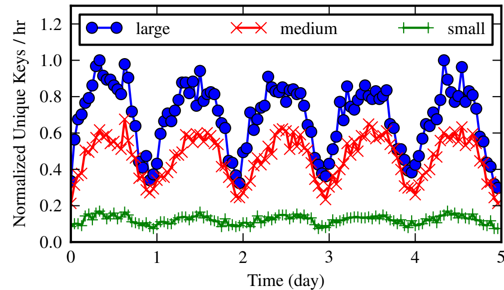
[3]We implicitly assume a FIFO ordering of data records over the network, as is typically the case with protocols like TCP.
[4]`http://www.akamai.com/dl/feature_sheets/Akamai_Download_Analytics.pdf`
[5]A beacon is simply an http `GET` issued by the download manager for a small GIF containing the reported values in its url query string.

Table 1: Queries used throughout the paper.

Name	Key	Value (aggregate type)	Description	Query Size
Small	`(cpid, bw)`	bytes downloaded (integer sum)	Total bytes downloaded by content provider by last-mile bandwidth.	$O(10^2)$ keys
Medium	`(cpid, bw, country_code)`	bytes downloaded (first 5 moments)	Mean and standard deviation of total bytes per download by content provider by bandwidth by country.	$O(10^4)$ keys
Large	`(cpid, bw, url)`	client ip (HyperLogLog)	Approximate number of unique clients by content provider by bw by url.	$O(10^6)$ keys



(a) CDF of the frequency per key at a single edge for the three queries.



(b) The unique key arrival rate for three different queries in a real-world web analytics service, normalized to the maximum rate for the large query.

Figure 3: Akamai Web download service data set characteristics.

three queries is the same, since each arriving beacon contributes a data record for each query. However, the three queries have a different distribution of those arrivals across the possible keys as shown in Figure 3(a). Recall that the large query has a large number of possible keys. About 56% of the keys for the large query arrived only once in the trace whereas the same percentage of keys for the medium and small query is 29% and 15% respectively. The median arrival rate per key was four (resp., nine) times larger for the medium (resp., small) query in comparison with the large query. Figure 3(b) shows the number of unique keys arriving per hour at an edge server for the three queries. The figure shows the hourly and daily variations and also the variation across the three queries.

4. MINIMIZING WAN TRAFFIC AND STALENESS

We now explore how to *simultaneously* minimize both traffic and staleness. We show that if the entire sequence of updates is known *beforehand*, then it is indeed possible to *simultaneously* achieve the optimal value for both traffic and staleness. While this offline solution is not implementable, it serves as a baseline to which any online algorithm can be compared. Further, it characterizes the optimal solution that helps us evolve the more sophisticated online algorithms that we present in Section 5.

Lemma 1 (Traffic optimality). *In each time window, an algorithm is* traffic-optimal *iff it flushes exactly one update to the center for each distinct key that arrived in the window.*

Proof. Any algorithm must flush at least one update for each distinct key that had arrivals in the time window. Suppose for contradiction that the algorithm flushes more than one update for a key. All flushes except the final one can be omitted, thereby decreasing the traffic, which is a contradiction. □

Intuitively, this lemma states that multiple records for each key within a window can be aggregated and sent as a single update to minimize traffic. Note that a pure batching algorithm satisfies the above lemma, and hence is traffic-optimal, but may not be staleness-optimal.

Lemma 2 (Staleness optimality). *Let the optimal staleness for a time window $[T-W,T)$ be S. For any $T-W \le t < T$, let $N(t)$ be the union of the set of keys that have outstanding updates (those not sent to the center yet) at time t and the set of keys that arrive in $[t,T)$. For a* staleness-optimal *algorithm the following holds:*

$$|N(t)| \le \int_t^{T+S} b(\tau)d\tau, \forall T-W \le t < T, \qquad (1)$$

where $b(\tau)$ is the instantaneous bandwidth at time τ. Further, if $S>0$ there exists a critical time t^* such that the above Inequality 1 is satisfied with an equality.

Proof. Inequality 1 holds since $N(t)$ records need to be flushed in interval $[t,T)$ and the maximum number of updates that can be sent before $T+S$ is $\int_t^{T+S} b(\tau)d\tau$. If $S>0$, then let t^* be the time of arrival of the last key in the time window. If $|N(t^*)| < \int_{t^*}^{T+S} b(\tau)d\tau$, for some $S' < S$, $|N(t^*)| = \int_{t^*}^{T+S'} b(\tau)d\tau$, since there are no new arrivals after t^*. Thus, all updates for keys in $N(t^*)$ can be

transmitted by time $T + S'$, decreasing the staleness to S', which is a contradiction. Hence, at time t^* Inequality 1 is satisfied with an equality. □

Intuitively, this lemma specifies that for a given arrival sequence, a staleness-optimal algorithm must send out pending updates to the center at a sufficient rate (dependent on the network bandwidth) to have them reach the center within the minimum feasible staleness bound.

Note that a pure streaming algorithm satisfies the above lemma, if the network has sufficient capacity to stream *all the arrivals* without causing network queues to build up. It, however, need not satisfy Lemma 1 if some of the groups have multiple arrivals within the window, and hence may not be traffic-optimal.

We now present *optimal offline algorithms* that minimize *both* traffic and staleness, provided the *entire* sequence of key updates is known to our aggregation algorithm *beforehand*.

THEOREM 3 (EAGER OPTIMAL ALGORITHM). *There exists an optimal offline algorithm that schedules its flushes* eagerly*; i.e., it flushes exactly one update for each distinct key immediately after the last arrival for that key within the time window.*

PROOF. Since the proposed algorithm flushes only a single update for each distinct key that arrived within the window, it is traffic-optimal as per Lemma 1. Clearly, any aggregation algorithm must flush an update for a key *after* the last arrival for that key, since the update must include the data contained in that last arrival in the final aggregate for that window. Suppose there exists a key where the last arrival was at time t but the update to the center was sent at $t+\delta$, for some $\delta > 0$. Modifying that schedule such that the update is flushed at time t instead of $t + \delta$ cannot increase staleness. Thus, there exists an eager schedule that achieves the same staleness. □

Intuitively, this algorithm is traffic-optimal since it sends only one update per key, and is also staleness-optimal since it sends each update without any additional delay. We call the optimal offline algorithm described above the *eager optimal algorithm* due to the fact that it eagerly flushes updates for each distinct key immediately after the final arrival to that key. This eager algorithm is just one possible algorithm to achieve both minimum traffic and staleness. It might well be possible to delay flushes for some groups and still achieve optimal traffic and staleness. An extreme version of such a scheduler is the *lazy optimal algorithm* that flushes updates at the last possible time that would still provide the optimal value of staleness and is described below.

1) Let keys $k_i, 1 \leq i \leq n$ have their last arrival at times $l_i, 1 \leq i \leq n$ respectively. Order the n keys that require flushing in the given window in the increasing order of their last update, i.e., the keys are ordered such that $l_1 \leq l_2 \leq \cdots l_n$.

2) Compute the minimum possible staleness S using the eager optimal algorithm.

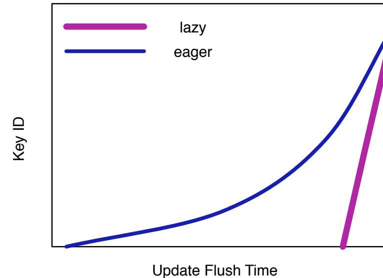


Figure 4: The lazy and eager algorithms define the extremes of a family of optimal algorithms.

3) As the base case, schedule the flush for the last key k_n at time $S - \delta_n$, where δ_n is the time required to transmit the update for k_n. That is, the last update is scheduled such that it arrives at the center with staleness exactly equal to S.

4) Now, iteratively schedule k_i, assuming all keys $k_j, j > i$ have already been scheduled. The update for k_i is scheduled at time $t_{i+1} - \delta_i$, where δ_i is the time required to transmit the update for k_i. That is, the update for k_i is scheduled such that update for k_{i+1} is scheduled immediately after the update of k_i completes.

THEOREM 4 (LAZY OPTIMAL ALGORITHM). *The lazy algorithm above is both traffic- and staleness-optimal.*

PROOF. By construction. □

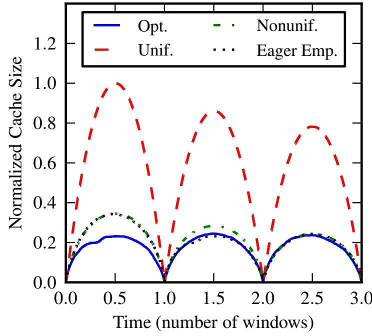
Further, consider a family of offline algorithms $\mathcal{A}$, where an algorithm $A \in \mathcal{A}$ schedules its update for key k_i at time t_i such that $e_i \leq t_i \leq l_i$, where e_i and l_i are the update times for key k_i in the eager and lazy schedules respectively. The following clearly holds.

THEOREM 5 (FAMILY OF OFFLINE OPTIMAL ALGORITHMS). *Any algorithm $A \in \mathcal{A}$ is both traffic- and staleness-optimal.*

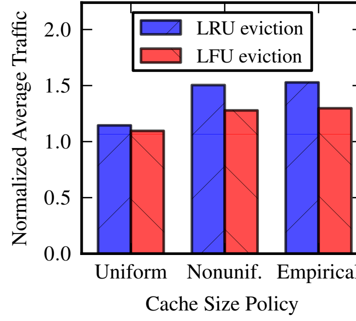
Figure 4 illustrates this concept. If each key is evicted from the cache no earlier than the eager eviction time (the left curve) and no later than the lazy eviction time (the right curve), then both staleness and traffic will be optimal.

5. PRACTICAL ONLINE ALGORITHMS

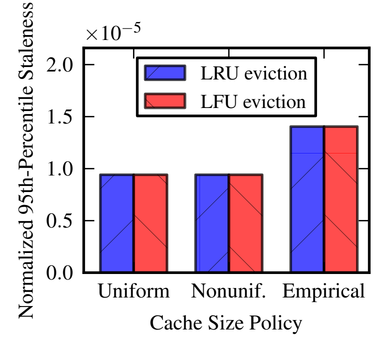
In this section, we explore practical online algorithms for grouped aggregation, that strive to minimize both traffic and staleness. To ease the design of such online algorithms, we first frame the edge aggregation problem as an equivalent *caching problem*. This formulation has two advantages. First, it allows us to decompose the problem into two subproblems: determining the *cache size*, and defining a *cache eviction policy*. Second, as we will show, while the first subproblem can be solved by using insights gained from the optimal offline algorithms, the second subproblem lends itself to using the enormous prior work on cache replacement policies [19].



(a) Cache sizes for emulations of the eager offline optimal algorithm.



(b) Traffic (normalized relative to an optimal algorithm)



(c) Staleness (normalized by window length)

Figure 5: Eager online algorithms.

Concretely, we frame the grouped aggregation problem as a caching problem by treating the set of aggregates $\{(k_i, \hat{v}_i)\}$ maintained at the edge as a cache. A novel aspect of our formulation is that the *size of this cache changes dynamically*. Concretely, the cache works as follows:

• *Cache insertion* occurs upon the arrival of a record (k, v). If an aggregate with key k and value v_e exists in the cache (a "cache hit"), the cached value for key k is updated as $v \oplus v_e$ where $\oplus$ is the binary aggregation operator defined in Section 2. If no aggregate exists with key k (a "cache miss"), then (k, v) is added to the cache.

• *Cache eviction* occurs as the result of a cache miss when the cache is already full, or due to a *decrease* in the cache size. When an aggregate is evicted, it is flushed downstream and cleared from the cache.

Given the above definition of cache mechanics, we can express any grouped aggregation algorithm as an equivalent caching algorithm where the key updates flushed by the aggregation algorithm correspond to key evictions of the caching algorithm. More formally:

THEOREM 6. *An aggregation algorithm A corresponds to a caching algorithm C such that:*

1. *At any time step, C maintains a cache size that equals the number of pending aggregates (those not sent to the center yet) for A, and*
2. *if A flushes an update for a key in a time step, C evicts the same key from its cache in that time step.*

Thus, any aggregation algorithm can be viewed as a caching algorithm with two policies: one for cache sizing and the other for cache replacement. In practice, we can define an online caching algorithm by defining online *cache size* and *cache eviction* policies. While the cache size policy determines *when* to send out updates, the cache eviction policy identifies *which* updates to send out at these times. Here we develop policies by attempting to emulate the behavior of the offline optimal algorithms using online information. We explore such online algorithms and the resulting tradeoffs in the rest of this section.

To evaluate the relative merits of these algorithms, we implement a simple simulator in Python. Our simulator models each algorithm as a function that maps from arrival sequences to update sequences. Traffic is simply the length of the update sequence, while staleness is evaluated by modeling the network as single-server queueing system with deterministic service times, and arrival times determined by the update sequence. Note that we have deliberately employed a simplified simulation, as the focus here is not on understanding performance in absolute terms, but rather to compare the tradeoffs between different algorithms. We use these insights to develop practical algorithms that we implement in Apache Storm and deploy on PlanetLab (Section 6).

Because we aim to emulate the offline optimal algorithms, it is useful to first study how these algorithms vary the cache size over the course of the window. Figure 6 shows, for three windows, the size of the cache for both eager and lazy optimal algorithms, for the `large` and `small` queries. Note that, for each query, cache sizes are normalized relative to the largest for that query.

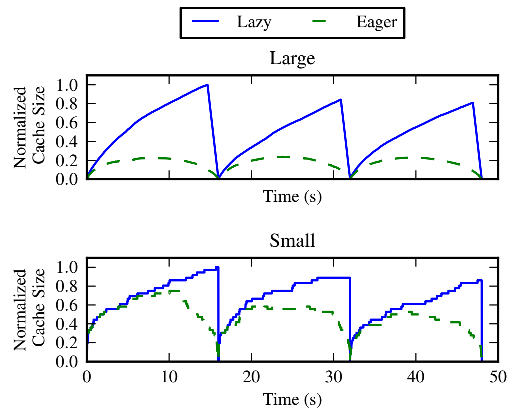


Figure 6: Cache size over time for eager and lazy offline optimal algorithms. Sizes are normalized relative to the largest size for the given query.

Several observations arise from this figure. First, we see that the eager algorithm maintains a smaller cache size than the lazy algorithm, as lazy retains some keys long after their last arrival. Second, the shape of the cache size under the eager algorithm is approximately symmetric about the center of the window, while the lazy algorithm allows the cache to grow until close to the end of the window, after which the size declines rapidly, at a nearly constant rate. Third, the time at which the lazy algorithm begins decreasing the cache sizes occurs earlier in the window for the larger query.

Armed with this high-level understanding of cache size behavior, we can begin developing practical online algorithms to emulate the offline optimal algorithms.

Note that throughout the remainder of this section, we present results for the `large` query due to space constraints, but similar trends also apply to the `small` and `medium` queries.

5.1 Emulating the Eager Optimal Algorithm

5.1.1 Cache Size

To emulate the cache size corresponding to that for an eager offline optimal algorithm, we observe that, at any given time instant, an aggregate for key k_i is cached only if: in the window, (i) there has already been an arrival for k_i, and (ii) another arrival for k_i is yet to occur. We attempt to compute the number of such keys using two broad approaches: analytical and empirical.

In our analytical approach, the eager optimal cache size at a time instant can be estimated by computing the *expected* number of keys at that instant for which the above conditions hold. To compute this value, we model the arrival process of records for each key k_i as a Poisson process with mean arrival rate λ_i. Then the probability $p_i(t)$ that the key k_i should be cached at a time instant t within a window $[T, T+W)$ is given by $p_i\left(t\right) = 1 - \hat{t}^{W\lambda_i} - \left(1-\hat{t}\right)^{W\lambda_i}$, where $\hat{t} = (t-T)/W$.[6]

We consider two different models to estimate the arrival processes for different keys. The first model is a *Uniform* analytical model, which assumes that key popularities are uniformly distributed, and each key has the same mean arrival rate λ. Then, if the total number of keys arriving during the window is k, the expected number of cached keys at time t is simply $k \cdot \left(1 - \hat{t}^{W\lambda} - \left(1-\hat{t}\right)^{W\lambda}\right)$.

However, as Figure 3(a) in Section 3 demonstrated, key popularities in reality may be far from uniform. A more accurate model is the *Nonuniform* analytical model, that assumes each key k_i has its own mean arrival rate λ_i, so that the expected number of cached keys at time t is given by $\sum_{i=1}^{k} p_i\left(\hat{t}\right)$.

An online algorithm built around these models requires predicting the number of unique keys k arriving during a window as well as their arrival rates λ_i. In our evaluation, we use a simple prediction: assume that the current window resembles the prior window, and derive these parameters from the arrival history in the prior window.

Our empirical approach, referred to as *Eager Empirical*, also uses the history from the prior window as follows: apply the eager offline optimal algorithm to the arrival sequence from the previous window, and use the resulting cache size at time $t - W$ as the prediction for cache size at time t.

Figure 5(a) plots the predicted cache size using these policies, along with the eager optimal cache size as a baseline. We observe that the Uniform model, unsurprisingly, is less accurate than the Nonuniform model. Specifically, it overestimates the cache size, as it incorrectly assumes that arrivals are uniformly distributed across many keys, rather than focused on a relatively small subset of relatively popular keys. Further, we see that the Eager Empirical model and the Nonuniform model both provide reasonably accurate predictions, but are prone to errors as the arrival rate changes from window to window.

5.1.2 Cache Eviction

Having determined the cache size, the next issue is which keys to evict when needed. We know that an optimal algorithm will only evict keys without future arrivals. However, determining such keys accurately requires knowledge of the future. Instead, to implement a practical online policy, we consider two popular practical eviction algorithms—namely least-recently used (LRU), and least-frequently used (LFU)—and examine their interaction with the above cache size policies.

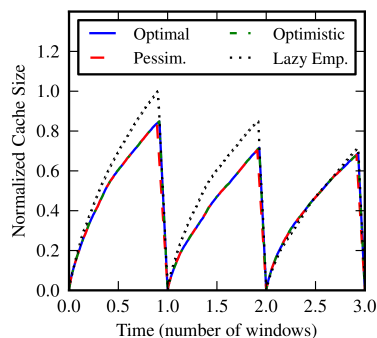
Figures 5(b) and 5(c) show the traffic and staleness, respectively, for different combinations of these cache size and cache eviction policies. Here, we simulate the case where network capacity is roughly five times that needed to support the full range of algorithms from pure batching to pure streaming. In these figures, traffic is normalized relative to the traffic generated by an optimal algorithm, while staleness is normalized by the window length.

From these figures, we see that the Eager Empirical and Nonuniform models yield similar traffic, though their staleness varies. It is worth noting that although the difference in staleness appears large in relative terms, the absolute values are still extremely low relative to the window length (less than 0.0015%), and are very close to optimal. We also see that LFU is the more effective eviction policy for this trace.

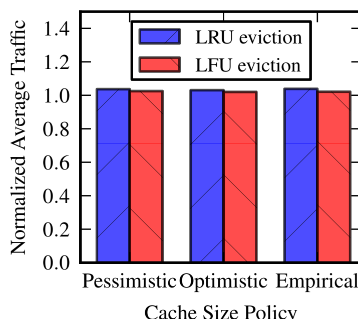
The more interesting result, however, is that the Uniform model, which produces the *worst* estimate of cache size, actually yields the *best* traffic: only about 9.6% higher than optimal, while achieving the same staleness as optimal. The reason is that, the more aggressive the cache size policy is in evicting keys prior to the end of the window, the more pressure it places on an imperfect cache eviction algorithm to predict which key is least likely to arrive again.

On the other hand, when combined with the most accurate model of eager optimal cache size (Nonuniform), even the best practical eviction policy (LFU) generates 28% more traffic than optimal. This result indicates that leaving more headroom in the cache size (as done by Uniform) provides more robustness to errors by an online cache eviction policy.

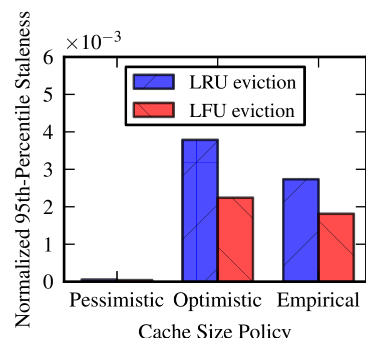
[6]Note that $W\lambda_i > 0$ since we are considering only keys with more than 0 arrivals, and that $\hat{t} < 1$ since $T \leq t < T + W$.



(a) Cache sizes for emulations of the lazy offline optimal algorithm.



(b) Traffic (normalized relative to an optimal algorithm)



(c) Staleness (normalized by window length)

Figure 7: Lazy online algorithms.

5.2 Emulating the Lazy Optimal Algorithm

5.2.1 Cache Size

To emulate the lazy optimal offline algorithm (Section 4), we estimate the cache size by working backwards from the end of the window, determining how large the cache should be such that it can be drained by the end of the window (or as soon as possible thereafter) by fully utilizing the network capacity. This estimation must account for the fact that new arrivals will still occur during the remainder of the window, and each of those that is a cache miss will lead to an additional update in the future. This leads to a cache size $c(t)$ at time t defined as: $c(t) = \max\left(\bar{b}\cdot(T-t) - M(t), 0\right)$, where $\bar{b}$ denotes the average available network bandwidth for the remainder of the window, T the end of the time window, and $M(t)$ the total number of cache misses that will occur during the remainder of the window.

Based on the above cache size function, an online algorithm needs to estimate the average bandwidth $\bar{b}$ and the number of cache misses $M(t)$ for the remainder of the window. We begin by focusing on the estimation of $M(t)$. We consider the bandwidth estimation problem in more detail in Section 5.3, and assume a perfect knowledge of $\bar{b}$ here. To estimate $M(t)$, we consider the following approaches. First, we can use a *Pessimistic* policy, where we assume that *all* remaining arrivals in the window will be cache misses. Concretely, we estimate $M(t) = \int_t^T a(\tau)\,\mathrm{d}\tau$ where $a(t)$ is the arrival rate at time t. In practice, this requires the prediction of the future arrival rate $a(t)$. In our evaluation, we simply assume that the future arrival rate is equal to the average arrival rate so far in the window.

Another alternative is to use an *Optimistic* policy, which assumes that the *current cache miss rate* will continue for the remainder of the window. In other words, $M(t) = \int_t^T m(\tau)a(\tau)\,\mathrm{d}\tau$ where $m(t)$ is the miss rate at time t. In our evaluation, we predict the arrival rate in the same manner as for the Pessimistic policy, and we use an exponentially weighted moving average for computing the recent cache miss rate.

A third approach is the *Lazy Empirical* policy, which is analogous to the Eager Empirical approach. It estimates the cache size by emulating the lazy offline optimal algorithm on the arrivals for the prior window.

Figure 7(a) shows the cache size produced by each of these policies. We see that both the Lazy Empirical and Optimistic models closely capture the behavior of the optimal algorithm in dynamically decreasing the cache size near the end of the window. The Pessimistic algorithm, by assuming that all future arrivals will be cache misses, decays the cache size more rapidly than the other algorithms.

5.2.2 Cache Eviction

We explore the same eviction algorithms here, namely LRU and LFU, as we did in Section 5.1.

Figures 7(b) and 7(c) show the traffic and staleness, respectively, generated by different combinations of these cache size and cache eviction policies. We see that LFU again slightly outperforms LRU. More importantly, we see that, regardless of which cache size policy we use, these lazy approaches outperform the best online eager algorithm in terms of traffic. Even the worst lazy online algorithm produces traffic less than 4% above optimal.

The results for staleness, however, show a significant difference between the different policies. We see that by assuming that all future arrivals will be cache misses, the Pessimistic policy achieves enough tolerance in the cache size estimation, avoiding overloading the network towards the end of the window, and leading to low staleness.

Based on the results so far, we see that accurately modeling the optimal cache size does not yield the best results in practice. Instead, our algorithms should be lazy, deferring updates until later in the window, and in choosing how long to defer, they should be pessimistic in their assumptions about future arrivals.

5.3 The Hybrid Algorithm

In the discussion of the lazy online algorithm above, we assumed perfect knowledge of the future network bandwidth $\bar{b}$. In practice, however, if the actual network capacity turns out to be lower than the predicted value, then too much traffic may back up close to the end of the window, potentially resulting in high staleness.

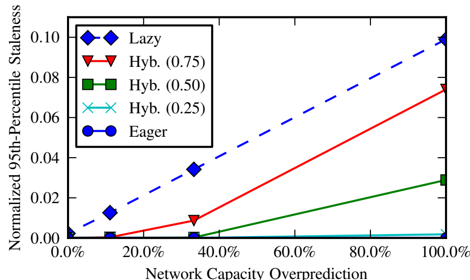


Figure 8: Sensitivity of the hybrid algorithms with a range of α values to overpredicting the available network capacity. Staleness is normalized by window length.

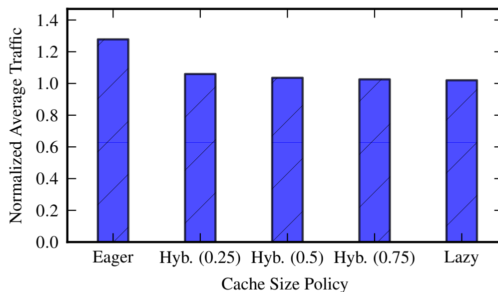


Figure 9: Average traffic for hybrid algorithms with several values of the laziness parameter α. Traffic is normalized relative to an optimal algorithm.

Figure 8 shows how staleness increases as the result of overpredicting network capacity. Note that the predicted capacity remains constant, while we vary the actual network capacity. The top-most curve corresponds to a lazy online algorithm (Pessimistic + LFU) which is susceptible to very high staleness if it overpredicts network capacity (up to 9.9% of the window length for 100% overprediction).

To avoid this problem, recall Theorem 5, where we observed that the eager and lazy optimal algorithms are merely two extremes in a family of optimal algorithms. Further, our results from Sections 5.1 and 5.2 showed that it is useful to add headroom to the accurate cache size estimates: towards a larger (resp., smaller) cache size in case of the eager (resp., lazy) algorithm. These insights indicate that a more effective cache size estimate should lie somewhere between the estimates for the eager and lazy algorithms. Hence, we propose a *Hybrid* algorithm that computes cache size as a linear combination of eager and lazy cache sizes. Concretely, a Hybrid algorithm with a *laziness parameter* α—denoted by Hybrid(α)—estimates the cache size $c(t)$ at time t as: $c(t) = \alpha \cdot c_l(t) + (1 - \alpha) \cdot c_e(t)$, where $c_l(t)$ and $c_e(t)$ are the lazy and eager cache size estimates, respectively. In our evaluation, we use the Nonuniform model for the eager and the Optimistic model for the lazy cache size estimation respectively, as these most accurately capture the cache sizes of their respective optimal baselines.

Observing Figure 8 again, we see that as we decrease the laziness parameter (α) below about 0.5, and use a more eager approach, the risk of bandwidth misprediction is largely mitigated, and the staleness even under significant bandwidth overprediction remains small.

Note that since predicted network capacity is constant in this figure, traffic is fixed for each algorithm irrespective of the bandwidth prediction error. Figure 9 shows that as we use a more eager hybrid algorithm, traffic increases. This illustrates a tradeoff between traffic and staleness in terms of achieving robustness to network bandwidth overprediction. A reasonable compromise seems to be a low α value, say 0.25. Using this algorithm, traffic is less than 6.0% above optimal, and even when network capacity is overpredicted by 100%, staleness remains below 0.19% of the window length.

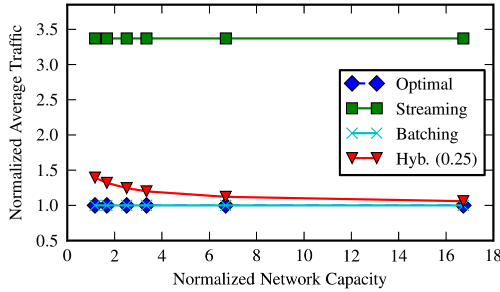
Overall, we find that a purely eager online algorithm is susceptible to errors by practical eviction policies, while a purely lazy online algorithm is susceptible to errors in bandwidth prediction. A hybrid algorithm that combines these two approaches provides a good compromise by being more robust to errors in both arrival process and bandwidth estimation.

5.4 Comparison to Optimal and Other Online Algorithms

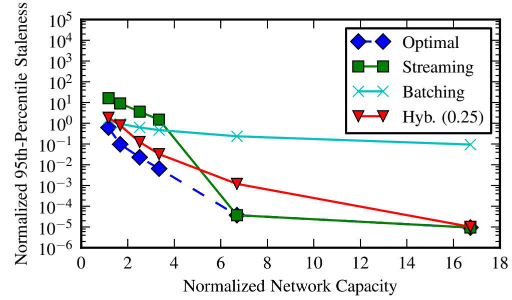
Finally, we compare our hybrid online aggregation algorithm against the simple aggregation algorithms of pure streaming and pure batching, as well as an offline optimal algorithm. So far, we have focused on a network regime of relatively high network capacity. To fully understand the tradeoffs of these algorithms, we now simulate these algorithms across a wide range of network capacities, ranging from highly constrained (less than 20% greater than optimal traffic) to highly unconstrained (about 5x more than that needed to support pure streaming).

Figures 10(a) and 10(b) show traffic and staleness, respectively, for the four algorithms over this range of network capacities. In terms of traffic, we see that our Hybrid(0.25) algorithm comes close to the optimal traffic, especially at higher network capacities. Traffic in the highly constrained regime is not as close to optimal, but still provides a significant improvement over that from pure streaming. The reason for this trend is that, as the network becomes more constrained, the envelope between lazy and eager algorithms shrinks, so that the hybrid algorithm has lower room for error. Note that batching is traffic-optimal, as discussed in Section 4.

In terms of staleness, we see that streaming goes from being nearly staleness-optimal for high network capacity to the worst when the capacity goes below a certain point. This is because under a highly constrained network, the excessive traffic from streaming leads to large (even unbounded) network queuing delays. The staleness for batching, on the other hand, goes from being the worst for high network capacity to close to optimal for low bandwidth. This is because it always defers communication until the end of the window, which can lead to high delay when network is not a bottleneck but prevents queue buildups under severe bandwidth constraints. Our Hybrid algorithm follows the same trend as the optimal, performing close to optimal irrespective of the network capacity.



(a) Traffic (normalized relative to an optimal algorithm)



(b) Staleness (normalized by window length). Y-axis is log-scale.

Figure 10: Traffic and staleness for different algorithms over a range of network capacities.

6. IMPLEMENTATION

We demonstrate the practicality of our algorithms and ultimately their performance by implementing them in Apache Storm [2]. Our prototype uses a distinct Storm cluster at each edge, as well as at the center, in order to distribute the work of aggregation. We choose this multi-cluster approach rather than attempting to deploy a single geo-distributed Storm cluster for two main reasons. First, a single global Storm cluster would require a custom task scheduler in order to control task placement. Second, and much more critically, Storm was designed and has been optimized for high performance within a single datacenter; it would not be reasonable to expect it to perform well in a geo-distributed setting characterized by high latency and high degrees of compute and bandwidth heterogeneity.

Figure 11 shows the overall architecture, including the edge and center Storm topologies. We briefly discuss each component in the order of data flow from edge to center.

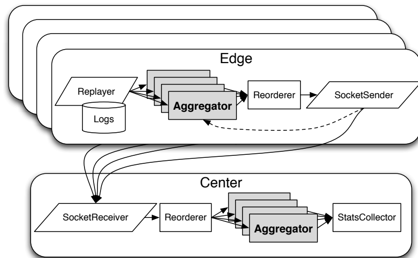


Figure 11: Aggregation is distributed over Apache Storm clusters at each edge as well as at the center.

6.1 Edge

Data enters our prototype through the `Replayer` spout. One instance of this spout runs within each edge, and it is responsible for replaying timestamped logs from a file, reproducing the original pattern of interarrival times. In order to allow us to explore different stream arrival rates, the `Replayer` takes a `speedupFactor` parameter, which dictates how much to speed up or slow down the log replay.

Each line of the logs is parsed using a query-specific parsing function, which produces a triple of `(timestamp, key, value)`. We leverage Twitter's Algebird[7] library to generalize over a broad class of aggregations, so the only restriction on value types is that they must have an Algebird `Semigroup` instance. This is already satisfied for many practical aggregations (e.g., integer sum, unique count via HyperLogLog, etc.), and implementing a custom `Semigroup` is straightforward. The `Replayer` emits these records downstream, and also periodically emits punctuation messages. Carrying only a timestmap, these punctuation messages simply denote that no messages with earlier timestamps will be sent in the future. The frequency of these punctuations is user-specified, though it is required that one be sent to mark the end of each time window.

The next step in the dataflow is the `Aggregator` bolt, for which one or more tasks run at each cluster. Each task is responsible for aggregating a hash-partitioned subset of the key space, and applying a cache size and eviction policy to determine when to transfer partial aggregates to the center. Each task maintains an in-memory key-value map, and uses the Algebird library to aggregate values for a given key. We generalize over a broad range of *eviction policies* by ordering keys using a priority queue with an efficient `changePriority` implementation, and consulting this priority queue to determine the next victim key when it becomes necessary. By defining priority as a function of key, value, existing priority (if any) and the time that the key was last updated in the map, we can capture a broad range of algorithms including LRU and LFU.

The `Aggregator` also maintains a cache size function, which maps from time within the window to a cache size. This function can be changed at runtime in order to support implementing arbitrary dynamic sizing policies. Specifically, a concrete `Aggregator` instance can install callback functions to be invoked upon the arrival of records or punctuations. This mechanism can be used, for example, to update the cache size function based on arrival rate and miss rate as in our Lazy Pessimistic algorithm, or to record the arrival history for one window and use this history to compute the size function for the next window, as in the Eager Empirical algorithm. For our experiments, we use this mechanism to implement a cache size policy that learns the eager optimal eviction schedule after processing the log trace once.

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

The `Aggregator` tasks send their output to a single instance of the `Reorderer` bolt. This bolt is responsible for delaying records as needed in order to maintain punctuation semantics. Data then flows into the `SocketSender` bolt, which connects to the central cluster at startup, and has the responsibility of serializing and transmitting partial aggregates downstream to the center using TCP sockets. This `SocketSender` also maintains an estimate of network bandwidth to the center, and periodically emits these estimates upstream to `Aggregator` instances for use in defining their cache size functions. Our bandwidth estimation is based on simple measurements of the rate at which messages can be sent over the network. For a more reliable prediction, we could employ lower-level techniques [7], or even external monitoring services [24].

6.2 Center

At the center, data follows largely the reverse order. First, the `SocketReceiver` spout is responsible for deserializing partial aggregates and punctuations and emitting 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 traffic and staleness, and these metrics are summarized by the final `StatsCollector` bolt.

Note that our prototype achieves at-most-once delivery semantics. Storm's acking mechanisms can be used to implement at-least-once semantics, and exactly-once semantics can be achieved by employing additional checks to filter duplicate updates, though we have not implemented these measures.

7. EXPERIMENTAL EVALUATION

To evaluate the performance of our algorithms in a real geo-distributed setting, we deploy our Apache Storm architecture on the PlanetLab testbed. Our PlanetLab deployment uses a total of eleven nodes (64 total cores) spanning seven sites. Central aggregation is performed using a Storm cluster at a single node at `princeton.edu`[8]. Edge locations include `csuohio.edu`, `uwaterloo.ca`, `yale.edu`, `washington.edu`, `ucla.edu`, and `wisc.edu`. Bandwidth from edge to center varies from as low as 4.5Mbps (`csuohio.edu`) to as high as 150Mbps (`yale.edu`), based on `iperf`. To simulate streaming data, each edge replays a geographic partition of the CDN log data described in Section 3. To explore the performance of our algorithms under a range of workloads, we use the three diverse queries described in Table 1, and we replay the logs at both low and high (8x faster than low) rates. Note that for confidentiality purposes, we do not disclose the actual replay rates, and we present staleness and traffic results normalized relative to the window length and optimal traffic, respectively.

[8] We originally employed multiple nodes at the center, but were forced to confine our central aggregation to a single node due to PlanetLab's restrictive limitations on daily network bandwidth usage that was quickly exhausted by the communication between Storm workers.

7.1 Aggregation using a Single Edge

Although our work is motivated by the general case of multiple edges, our algorithms were developed based on an in-depth study of the interaction between a single edge and center. We therefore begin by studying the real-world performance of our hybrid algorithm when applied at a single edge. Following the rationale from Section 5.3, we choose a laziness parameter of $\alpha = 0.25$ for this initial experiment, though we will study the tradeoffs of different parameter values shortly.

Compared to the extremes of pure batching and pure streaming, as well as an optimal algorithm based on a priori knowledge of the data stream, our algorithm performs quite well. Figures 12(a) and 12(b) show that our hybrid algorithm very effectively exploits the opportunity to reduce bandwidth relative to streaming, yielding traffic less than 2% higher than the optimal algorithm. At the same time, our hybrid algorithm is able to reduce staleness by 65% relative to a pure batching algorithm.

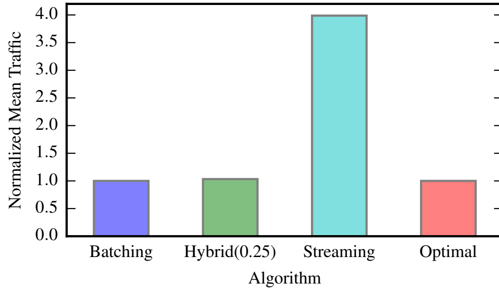
7.2 Scaling to Multiple Edges

Now, in order to understand how well our algorithm scales beyond a single edge, we partition the log data over three geo-distributed edges. We replay the logs at both low and high rates, and for each of the `large`, `medium`, and `small` queries[9]. As Figures 13(a) and 13(b) demonstrate, our hybrid algorithm performs well throughout. It is worth noting that the edges apply their cache size and cache eviction policies based purely on local information, without knowledge of the decisions made by the other edges, except indirectly via the effect that those decisions have on the available network bandwidth to the edge.

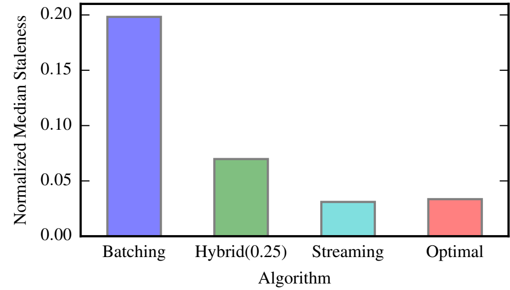
Performance is generally more favorable for our algorithm for the `large` and `medium` queries than for the `small` query. The reason is that, for these larger queries, while edge aggregation reduces communication volume, there is still a great deal of data to transfer from the edges to the center. Staleness is quite sensitive to precisely when these partial aggregates are transferred, and our algorithms work well in scheduling this communication. For the `small` query, on the other hand, edge aggregation is extremely effective in reducing data volumes, so much so that there is little risk in delaying communication until the end of the window. For queries that aggregate extremely well, batching is a promising algorithm, and we do not necessarily outperform batching. The advantage of our algorithm over batching is therefore its broader applicability: the Hybrid algorithm performs roughly as well as batching for small queries, and significantly outperforms it for large queries.

We continue by further partitioning the log data across a total of six geo-distributed edges. Given the higher aggregate compute and network capacity of this deployment, we focus on the `large` query at both low and high arrival rates. From Figure 14(a), we can again observe that our hybrid algorithm yields near-optimal traffic. We can also observe an important effect of stream arrival rate: all else equal, a high

[9] We do not present the results for `large`-high because the amount of traffic generated in these experiments could not be sustained within the PlanetLab bandwidth limits.

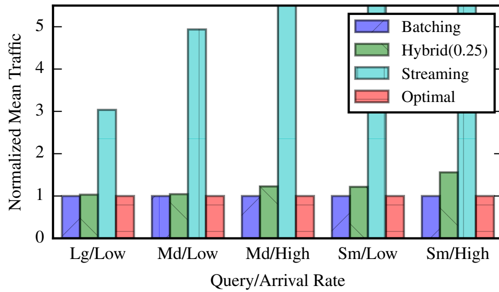


(a) Mean traffic (normalized relative to an optimal algorithm).

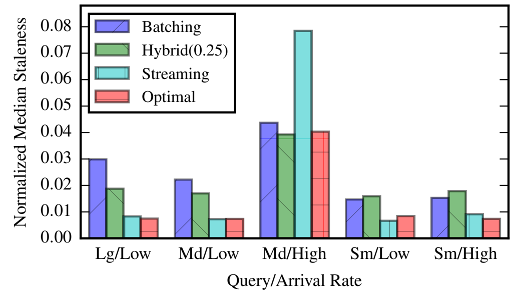


(b) Median staleness (normalized by window length).

Figure 12: Performance for batching, streaming, optimal, and our hybrid algorithm for the `large` query with a low stream arrival rate using a one-edge Apache Storm deployment on PlanetLab.



(a) Mean traffic (normalized relative to an optimal algorithm). Normalized traffic values for streaming are truncated, as they range as high as 164.



(b) Median staleness (normalized by window length).

Figure 13: Performance for batching, streaming, optimal, and our hybrid algorithm for a range of queries and stream arrival rates using a three-edge Apache Storm deployment on PlanetLab.

stream arrival rate lends itself to more thorough aggregation at the edge. This is evident in the higher normalized traffic for streaming with the high arrival rate than with the low arrival rate.

In terms of staleness, Figure 14(b) shows that our algorithm performs well for the high arrival rate, where the network capacity is more highly constrained, and staleness is therefore more sensitive to the particular scheduling algorithm. At the low arrival rate, we see that our hybrid algorithm performs slightly worse than batching, though in absolute terms this difference is quite small. Our hybrid algorithm generates higher staleness than streaming, but does so at a much lower traffic cost. Just as with the three-edge case, we again see that, where a large opportunity exists, our algorithm exploits it, and where an extreme algorithm such as batching already suffices, our algorithm remains competitive.

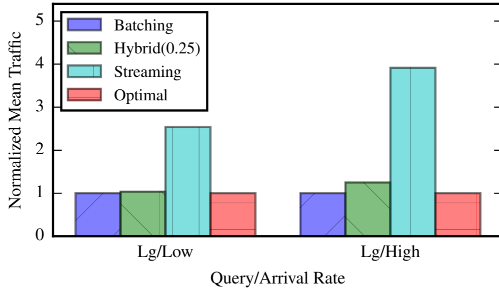
7.3 Effect of Laziness Parameter

In Section 5.3, we observed that a purely eager algorithm is vulnerable to mispredicting which keys will receive further arrivals, while a purely lazy algorithm is vulnerable to overpredicting network bandwidth. This motivated our hybrid algorithm, which uses a linear combination of eager and lazy cache size functions. We explore the real-world tradeoffs of using a more or less lazy algorithm by running experiments with the `large` query at a low replay rate over three edges with laziness parameter α ranging from 0 through 1.0 by steps of 0.25. As expected based on our simulation results, Figure 15(a) shows that α has little effect on traffic when it exceeds about 0.25. Somewhere below this value, the imperfections of practical cache eviction algorithms (LRU in our implementation) begin to manifest. More specifically, at $\alpha = 0$, the hybrid algorithm reduces to a purely eager algorithm, which makes eviction decisions well ahead of the end of the window, and often chooses the wrong victim. By introducing even a small amount of laziness, say with $\alpha = 0.25$, this effect is largely mitigated.

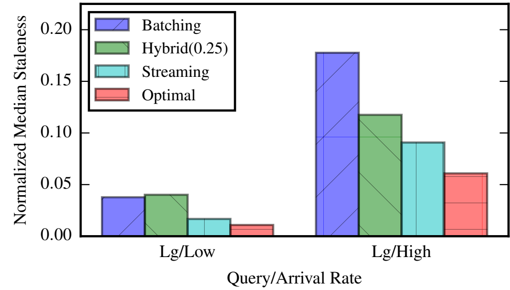
Figure 15(b) shows the opposite side of this tradeoff: a lazier algorithm runs a higher risk of deferring communication too long, in turn leading to higher staleness. Based on staleness alone, a more eager algorithm is better. Based on the shape of these trends, we have chosen to use $\alpha = 0.25$ throughout our experiments, but this may not be the optimal value. Further study would be necessary to determine an optimal value of α, and this optimal choice may in fact depend on the relative importance of minimizing staleness versus minimizing traffic.

8. RELATED WORK

Aggregation: Aggregation is a key operator in analytics, and grouped aggregation is supported by many data-parallel programming models [8, 11, 25]. Larson et al. [15] explore the benefits of performing partial aggregation prior to a join

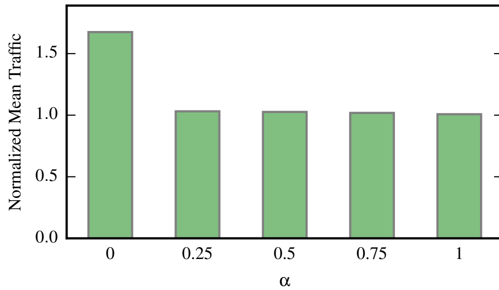


(a) Mean traffic (normalized relative to an optimal algorithm).

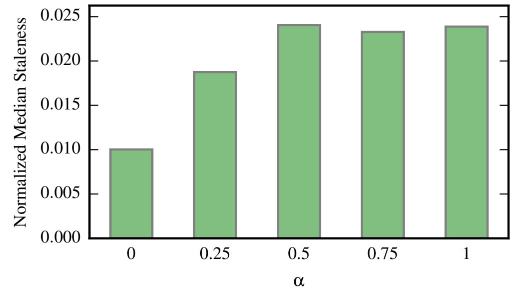


(b) Median staleness (normalized by window length).

Figure 14: Performance for batching, streaming, optimal, and our hybrid algorithm for the `large` query with low and high stream arrival rates using a six-edge Apache Storm deployment on PlanetLab.



(a) Mean traffic (normalized relative to an optimal algorithm).



(b) Median staleness (normalized by window length).

Figure 15: Effect of laziness parameter α using a three-edge Apache Storm deployment on PlanetLab with query `large`.

operation, much as we do prior to network transmission. While they also recognize similarities to caching, they consider only a fixed-size cache, whereas our approach uses a dynamically varying cache size. In sensor networks, aggregation is often performed over a hierarchical topology to improve energy efficiency and network longevity [16, 22], whereas we focus on cost (traffic) and information quality (staleness). Amur et al. [6] study grouped aggregation, focusing on the design and implementation of efficient data structures for batch and streaming computation. They discuss tradeoffs between eager and lazy aggregation, but do not consider the effect on staleness, a key performance metric in our work.

Streaming systems: Numerous streaming systems [5, 9, 20, 26] 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 achieve high quality results (low staleness) at low cost (low traffic).

Wide-area computing: 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 [13, 23]. Relatively little work on streaming computation [3] has focused on wide-area deployments, or associated questions such as where to place computation. Pietzuch et al. [18] optimize operator placement in geo-distributed settings to balance between system-level bandwidth usage and latency. Hwang et al. [14] rely on replication across the wide area in order to achieve fault tolerance and reduce straggler effects. JetStream [21] considers wide-area streaming computation, but unlike our work, assumes that it is always better to push more computation to the edge.

Optimization tradeoffs: LazyBase [10] provides a mechanism to trade off increased staleness for faster query response in the case of ad-hoc queries. BlinkDB [4] and JetStream [21] provide mechanisms to trade off accuracy with response time and bandwidth utilization, respectively. We focus on *jointly* optimizing both network traffic and staleness.

9. CONCLUSION

In this paper, we focused on optimizing the important primitive of windowed grouped aggregation in a wide-area streaming analytics setting on two key metrics: WAN traffic and staleness. We presented a family of optimal offline algorithms that *jointly minimize both staleness and traffic*. Using this as a foundation, we developed practical online aggregation algorithms based on the observation that grouped aggregation can be modeled as a *caching problem* where the cache size varies over time. We explored a range of online algorithms ranging from eager to lazy in terms of how soon they send out updates. We found that a hybrid online algorithm works best in practice, as it is robust to a wide range of network constraints and estimation errors. We demonstrated the practicality of our algorithms through an implementation in Apache Storm, deployed on the PlanetLab testbed. The results of our experiments, driven by workloads

derived from anonymized traces of Akamai's web analytics service, showed that our online aggregation algorithms perform close to the optimal algorithms for a variety of system configurations, stream arrival rates, and query types.

10. ACKNOWLEDGMENTS

The authors would like to thank Ravali Kandur for her help deploying Apache Storm on PlanetLab. They would also like to acknowledge NSF grant CNS-1413998 which supported this research. This work was also supported in part by an IBM Faculty Award.

11. REFERENCES

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

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

[3] D. J. Abadi et al. The design of the borealis stream processing engine. In *Proc. of CIDR*, pages 277–289, 2005.

[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] H. Amur et al. Memory-efficient groupby-aggregate using compressed buffer trees. In *Proc. of SoCC*, 2013.

[7] J. Bolliger and T. Gross. Bandwidth monitoring for network-aware applications. In *Proc. of HPDC*, pages 241–251, 2001.

[8] 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.

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

[10] J. Cipar, G. Ganger, K. Keeton, C. B. Morrey, III, C. A. Soules, and A. Veitch. LazyBase: trading freshness for performance in a scalable database. In *Proc. of EuroSys*, pages 169–182, 2012.

[11] 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.

[12] B. Heintz, A. Chandra, and R. K. Sitaraman. Towards optimizing wide-area streaming analytics. In *Proc. of the 2nd IEEE Workshop on Cloud Analytics*, 2015.

[13] B. Heintz, A. Chandra, R. K. Sitaraman, and J. Weissman. End-to-end optimization for geo-distributed mapreduce. *IEEE TCC*, 2015.

[14] 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.

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

[16] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A Tiny AGgregation service for ad-hoc sensor networks. In *Proc. of OSDI*, 2002.

[17] 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.

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

[19] S. Podlipnig and L. Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, Dec. 2003.

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

[21] 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.

[22] R. Rajagopalan and P. Varshney. Data-aggregation techniques in sensor networks: A survey. *IEEE Communications Surveys Tutorials*, 8(4):48–63, 2006.

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

[24] R. Wolski, N. T. Spring, and J. Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Gener. Comput. Syst.*, 15(5-6):757–768, Oct. 1999.

[25] 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.

[26] 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.