This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2017.2750678, IEEE Transactions on Cloud Computing

Optimizing Timeliness and Cost in Geo-Distributed Streaming Analytics

Benjamin Heintz, *Member, IEEE,* Abhishek Chandra, *Member, IEEE,* Ramesh K. Sitaraman, *Senior Member, IEEE*

Abstract—Rapid data streams are generated continuously from diverse sources including 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 geo-distributed analytics service uses a hub-and-spoke model, comprising multiple edges connected by a wide-area-network (WAN) to a central data warehouse. In this paper, we focus on the widely used primitive of *windowed grouped aggregation*, and examine the question of *how much computation should be performed at the edges versus the center*. We develop algorithms to optimize two key metrics: *WAN traffic* and *staleness* (delay in getting results). We present a family of optimal offline algorithms that *jointly minimize* these metrics, and we use these to guide our design of practical online algorithms based on the insight that windowed grouped aggregation can be modeled as a *caching* problem where the cache size varies over time. We evaluate our algorithms through an implementation in Apache Storm deployed on PlanetLab. Using workloads derived from anonymized traces of a popular analytics service from a large commercial CDN, our experiments show that our online algorithms achieve near-optimal traffic and staleness for a variety of system configurations, stream arrival rates, and queries.

Index Terms—Geo-distributed data analytics, Stream computing, Scheduling, Resource management, Windowed aggregation.

1 Introduction

Data analytics is undergoing a revolution: the volume and velocity of data sources are increasing rapidly. Across a number of application domains from web, social, and energy analytics to scientific computing, large quantities of data are generated continuously in the form of posts, tweets, logs, sensor readings, and more. A modern analytics service must provide real-time analysis of these data streams to extract meaningful and timely information. As a result, there has been a growing interest in streaming analytics with recent development of several distributed analytics platforms [1], [2], [3].

In many streaming analytics domains, inputs originate from diverse sources including 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. Data sources send streams of data to nearby "edge" servers. These geographically distributed edge servers process incoming data and send summaries to a central location that can process the data further, store summaries, and present those summaries in visual form to users of the analytics service. While the central hub is typically located in a well-provisioned data center, resources may be limited at the edge locations. In particular, the available WAN bandwidth between the edge and the center is 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. This approach is generally suboptimal, because it strains the scarce WAN bandwidth available between the edge and the center, leading to delayed results. Further, it fails to make use of the available compute and storage resources at the edge. An alternative is a *decentralized approach* [4] that utilizes the edge for much of the processing in order to minimize 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*. Abstractions for grouped aggregation are provided in most data analytics frameworks, for example as the `Reduce` operation in MapReduce, or `Group By` in SQL and LINQ. A useful variant in stream computing is *windowed* grouped aggregation, where each group is further broken down into finite time windows before being 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 her web site broken down by country and aggregated on an hourly basis to gauge the current content popularity. A network operator may need to compute the average load in different parts of the network every 5 minutes to identify hotspots. In these cases, users define a standing windowed grouped aggregation query that generates results periodically for each time window

- *B. Heintz was with the Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455. He is now with Facebook, Menlo Park, CA 94025. E-mail: heintz@cs.umn.edu*
- *A. Chandra is with the Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455. E-mail: chandra@cs.umn.edu*
- *R. K. Sitaraman is with the Department of Computer Science, University of Massachusetts, Amherst, MA 01003, and Akamai Technologies. E-mail: ramesh@cs.umass.edu*

2168-7161 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2017.2750678, IEEE Transactions on Cloud Computing

(every hour, 5 minutes, etc.).

Our work focuses 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). A service provider typically pays for the WAN bandwidth used by its deployed servers [5], [6]. In fact, bandwidth cost incurred due to WAN traffic is a significant component of the operating expense (OPEX) of the service provider infrastructure, the other key components being colocation and power costs. Therefore reducing WAN traffic represents an important cost-saving opportunity. In addition, reducing staleness is critical in order to deliver timely results to applications and is often a part of the SLA for analytics services. While much of the existing work on decentralized analytics [4], [7] 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 and improved timeliness. The key decision that our algorithms make is *how much of the data aggregation should be performed at the edge versus the center*.

1.1 Challenges in optimizing grouped aggregation

Consider two alternate approaches to grouped aggregation: *pure streaming*, where all data is immediately sent from edge to 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. Let us consider simplified examples of two queries from our web analytics application exemplar to illustrate the complexities. For each example below, let us assume that the edge can communicate at the rate of $1{,}000$ records per second to the center.

Example 1: Consider a query that uses grouped aggregation with key-value pairs $(\mathrm{key} = \mathrm{country}, \mathrm{value} = \mathrm{visits})$ for a time window of one hour. We say this query is "small" since the number of possible distinct keys is small as there are no more than 200 countries in the world. For this query, the right approach might be a pure batching approach. This approach will minimize the WAN traffic, since at most one record is sent per distinct key. Further, the staleness of the final result might also be small; there are at most 200 records to send so the aggregate values can be sent to the center within $200/1{,}000 = 0.2$ seconds of the end of the window.[1]

Example 2: Consider a second query that uses grouped aggregation with key-value pairs $(\mathrm{key} = \langle \mathrm{userid}, \mathrm{url} \rangle, \mathrm{value} = \mathrm{visits})$ for a time window of one hour. We say that this query is "large" since the product of the unique users and urls for a large website, and hence, the number of possible distinct keys received within a time window, could be large. Let us assume this number to be 100,000 distinct keys per window. The above batching solution may not work well for the larger query, since sending 100,000 records at the end of the time window can incur a delay of 100 seconds, resulting in higher staleness. A more appropriate approach might be pure streaming. Since each unique user/url combination is less likely to repeat within a time window, the extra traffic sent by streaming is relatively small. Further, there is no backlog of keys that must be updated at the end of the time window, resulting in smaller staleness.

These examples illustrate that the right strategy for optimizing grouped aggregation depends on the data and query characteristics, including rate of key arrivals at the edge, the number of unique keys seen in a time window, the available edge-to-center bandwidth, etc. To show that the intuition gained from these examples is true in practice, we ran experiments on a 5-node PlanetLab [8] setup using the traces of a popular web analytics service offered by a large CDN (see Section 3 for details on the dataset and queries used in our experiments). Figure 1 shows the traffic and staleness obtained for different queries (small, medium and large) obtained for the pure streaming and pure batching algorithms. The figure shows that, depending on the query, pure batching could reduce traffic by 70-90% while increasing staleness by 37-300% relative to pure streaming. This illustrates that simple aggregation algorithms cannot optimize both traffic and staleness at the same time. Instead they optimize one key metric at the cost of the other. Further, the factors influencing the performance of such algorithms are often hard to predict and can vary significantly over time—see Figure 2(b)—requiring the design of algorithms that can adapt to changing factors in a dynamic fashion.

1.2 Research contributions

• To our knowledge, we provide the first algorithms and analysis for optimizing geo-distributed windowed grouped aggregation. In particular, we show that simple 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 windowed grouped aggregation can be modeled as a *caching problem* where the cache size varies over time. This 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 [1], deployed on the PlanetLab [8] testbed. Our experiments are driven by workloads derived from anonymized traces of a popular web analytics service offered by Akamai [9], a large content delivery network. The results of our experiments show that our online aggregation algorithms simultaneously achieve traffic less than 2% higher than 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.

• We show how our algorithms can be extended to a *hopping window*-based grouped aggregation problem, and explore the tradeoffs w.r.t. the relative size of the window and hopping duration.

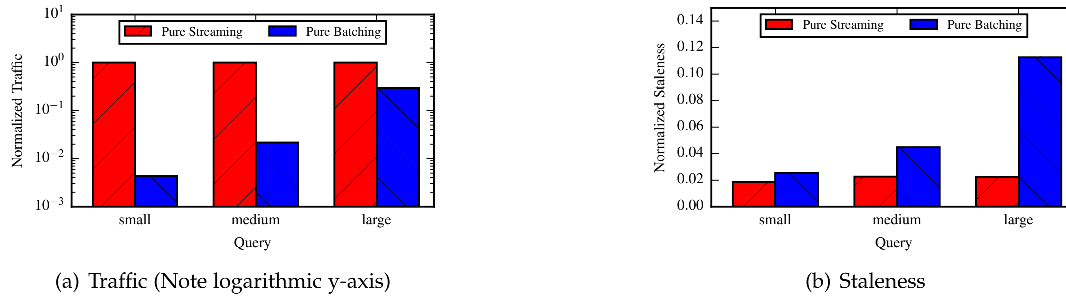
2 Problem Formulation

System model: We consider the typical hub-and-spoke architecture of an analytics system with a center and multiple

[1] WAN *latency* also contributes to delay, but because it is a function of the underlying network infrastructure, it is not something we can directly control through our algorithms. We therefore focus our attention on network delays due to wide-area *bandwidth* constraints.

2168-7161 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2017.2750678, IEEE Transactions on Cloud Computing



(a) Traffic (Note logarithmic y-axis)

(b) Staleness

Fig. 1. Simple aggregation algorithms such as streaming and batching trade off one metric for the other, and their performance depends on data and query characteristics.

edges. Data streams are first sent from each source to a nearby edge. The edges collect and (potentially, partially) aggregate the data. The aggregated data can then be sent from the edges to the center for final aggregation. 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 with key k, an aggregate value $\hat{v} = v_1 \oplus v_2 \cdots \oplus v_n$ is computed, where $\oplus$ is an application-defined associative binary operator; e.g., `Sum`, `Max`, or `HyperLogLog` merge.[2] Customarily, the timeline is subdivided into non-overlapping intervals of size W and grouped aggregation is computed on each such interval. We consider such non-overlapping time windows (often called *tumbling* windows in analytics terminology) for much of the paper, but consider a generalization to overlapping windows called *hopping* windows in Section 8.

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 measure of timeliness; and *network traffic*, a 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 $T + W + s$. WAN traffic is measured by the number of updates sent over the network from the edge to 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[3] 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 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.[4]

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[5] 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 performance the users are experiencing, how many downloads complete successfully, and so on. The data source is a software called Download Manager that is installed on mobile devices, laptops, and desktops of millions of

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

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

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

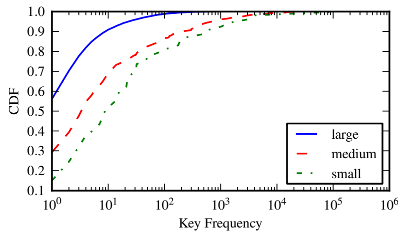
[5]https://goo.gl/QuKRkD

2168-7161 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

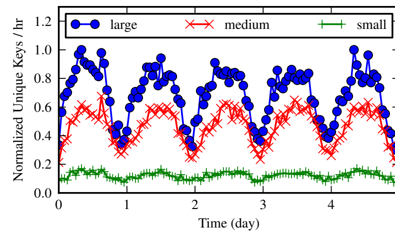
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2017.2750678, IEEE Transactions on Cloud Computing

TABLE 1
Queries used throughout the paper.

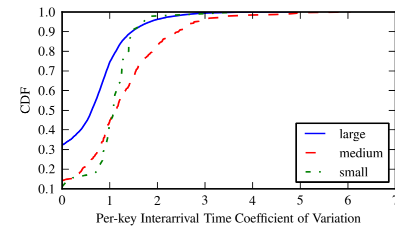
Name	Key	Value (aggregate type)	Description	Query Size
Small	(cpid, bw)	bytes downloaded (integer sum)	Total bytes downloaded by content provider by last-mile bandwidth.	$O(10^2)$ keys
Medium	(cpid, bw, country_code)	bytes downloaded (first 5 moments)	Mean and standard deviation of total bytes per download 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.



(c) CDF of the coefficient of variation of per-key interarrival times, for keys with at least 25 arrivals.

Fig. 2. Akamai Web download service data set characteristics.

users around the world and is used to download software updates, security patches, music, games, and other content. Each Download Manager instance logs information about its downloads to the widely-deployed edge servers using "beacons". These beacons contain anonymized information about the download start time, the 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 common queries in the download analytics service. These queries can be roughly classified according to *query size*, defined as 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 a key comprising the content provider id and the user's last mile bandwidth classified into four buckets. The `medium` query groups by content provider id, user's last mile bandwidth, and the user's country code. The `large` query groups by content provider id, the user's country code, and the url accessed. Note that the last dimension—url—can assume 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 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 arrivals across keys as shown in Figure 2(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 corresponding 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 than for the large query. Figure 2(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. Figure 2(c) shows the distribution of variation in interarrival times. Many keys have a coefficient of variation greater than 1, indicating relatively bursty arrivals.

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 arrivals is known *beforehand*, then it is indeed possible to simultaneously achieve both optimal traffic and optimal 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 develop the 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. □

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 $[T, T+W)$ be S. For any $T \leq t < T+W$, 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*

2168-7161 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

that arrive in $[t, T+W)$. *For a* staleness-optimal *algorithm the following holds:*

$$|N(t)| \leq \int_{t}^{T+S} b(\tau)d\tau, \forall T \leq t < T+W, \quad (1)$$

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 $[t, T+W)$ and the maximum number of updates that can be sent before $T+W+S$ is $\int_t^{T+W+S} b(\tau)d\tau$. If $S > 0$, then let t^* be the time of arrival of the last key in the time window. If $|N(t^*)| < \int_{t^*}^{T+W+S} b(\tau)d\tau$, for some $S' < S$, $|N(t^*)| = \int_{t^*}^{T+W+S'} b(\tau)d\tau$, since there are no new arrivals after t^*. Thus, all updates for keys in $N(t^*)$ can be transmitted by time $T+W+S'$, 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 Lemma 2 if the network has sufficient capacity to stream *all arrivals* without causing network queues to grow. It need not, however, satisfy Lemma 1 if some 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 arrivals is known 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 (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. 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. □

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 is possible to delay flushes for some groups and still achieve optimal traffic and staleness. An extreme version of such an algorithm is a *lazy optimal algorithm* 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 in increasing order of their last arrival, 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.
3) As the base case, schedule the flush for 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.

Intuitively, this algorithm uses the eager offline optimal algorithm to determine the optimal value of staleness S for a window, and schedules its updates to start at the last possible time that would still enable it to flush all its updates with a staleness of S. It is traffic-optimal because it flushes an update for a key only after the last arrival for that key.

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, because A never sends more traffic than the eager optimal algorithm, and its staleness cannot be worse than that of the lazy optimal algorithm.

Theorem 5 (Family of Offline Optimal Algorithms)**.** *Any algorithm* $A \in \mathcal{A}$ *is both traffic- and staleness-optimal.*

Proof. By construction. □

5 PRACTICAL ONLINE ALGORITHMS

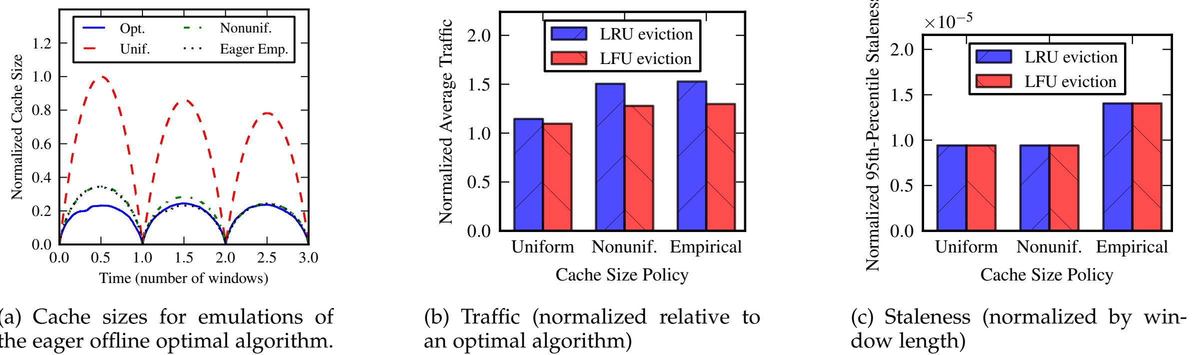
In this section, we explore practical online algorithms for grouped aggregation, that strive to minimize both traffic and staleness. To ease the design of such online algorithms, we first frame the edge aggregation problem as an equivalent *caching problem*. This formulation has two advantages. First, 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 vast prior work on cache replacement policies [10].

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

2168-7161 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2017.2750678, IEEE Transactions on Cloud Computing



(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)

Fig. 3. Eager online algorithms.

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 keys flushed by the aggregation algorithm correspond to keys evicted by 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.*

Proof. By construction. □

Thus, any aggregation algorithm can be viewed as a caching algorithm with two policies: one for cache sizing and the other for cache eviction. 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 mapping from arrival sequences to update sequences. Traffic is simply the length of the update sequence, while staleness is evaluated by modeling the network as a 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).

Note that throughout 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 of an eager offline optimal algorithm, we observe that, at any given time, 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 $p_i(t) = 1 - \hat{t}^{W\lambda_i} - \left(1-\hat{t}\right)^{W\lambda_i}$, 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 2(a) demonstrated, key popularities in reality may be far from uniform. A more accurate model is the *Nonuniform* analytical model, which 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.

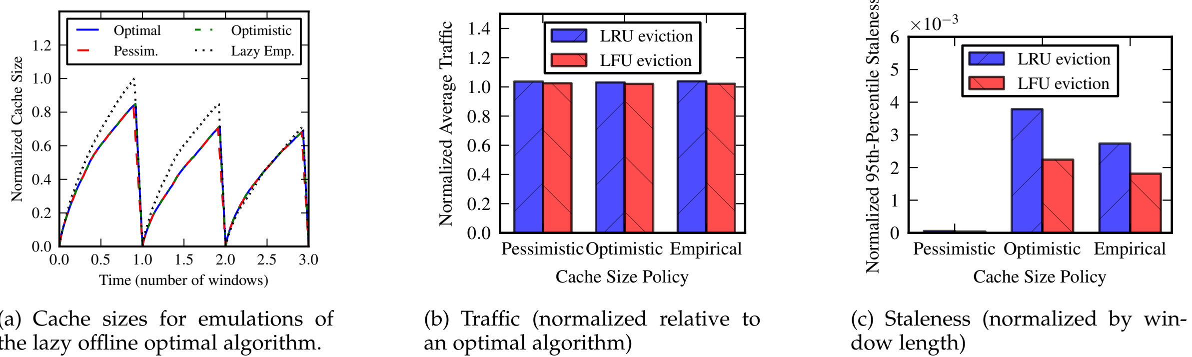
Our empirical approach, referred to as *Eager Empirical*, also uses the history from the prior window as follows: apply the eager offline optimal algorithm to the arrival sequence from the previous window, and use the resulting cache size at time $t-W$ as the cache size at time t.

Figure 3(a) plots the cache size using these policies, along with the eager optimal size as a baseline. We observe that the Uniform model, unsurprisingly, is less accurate than the

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

2168-7161 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2017.2750678, IEEE Transactions on Cloud Computing



(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)

Fig. 4. Lazy online algorithms.

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 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

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 3(b) and 3(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.

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, $M(t) = m(t)\int_t^T a(\tau)\,\mathrm{d}\tau$ 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 to track the 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 4(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

2168-7161 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2017.2750678, IEEE Transactions on Cloud Computing

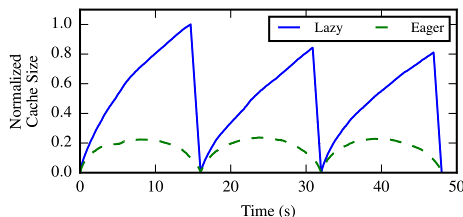


Fig. 5. Cache size over time for eager and lazy offline optimal algorithms. Sizes are normalized relative to the largest size.

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 4(b) and 4(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 cache size policy, 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 is lower than predicted, then network queueing delays may grow, leading to high staleness. For example, a lazy online algorithm (Pessimistic + LFU) can lead to a staleness of up to 9.9% of the window length for 100% overprediction.

To avoid this problem, recall Theorem 5, which states that the eager and lazy optimal algorithms are merely two extremes in a family of algorithms. Figure 5 shows, for three windows, the size of the cache for both eager and lazy optimal algorithms. We observe that the eager algorithm maintains a smaller cache size than the lazy algorithm, as lazy retains some keys long after their last arrival. 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.

Our simulation results (omitted here due to space constraints) show 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.

There is, however, a tradeoff: as we use a more eager hybrid algorithm, traffic increases. A low α value, say 0.25, provides a reasonable compromise. 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 of these two algorithms provides a good compromise by being more robust to errors in both arrival process and bandwidth estimation.

6 Implementation

We demonstrate the practicality of our algorithms by implementing them in Apache Storm [1]. Our prototype uses a distinct Storm cluster at each edge, as well as at the center, in order to distribute the work of aggregation, and to emulate a geo-distributed hub-and-spoke infrastructure.

Edge: Data enters our prototype at the edge through the `Replayer` spout, which replays timestamped logs from a file, and can speed up or slow down log replay to explore different stream arrival rates. Each line is parsed using a query-specific parsing function to produce a `(timestamp, key, value)` triple. Our implementation supports a broad set of value types and associated aggregations by leveraging Twitter's Algebird[7] library. The `Replayer` emits records according to their timestamp (i.e., event time and processing time [11] are equivalent at the `Replayer`) and also periodically emits punctuation messages to indicate that no messages with earlier timestamps will be sent in the future.

The next step in the dataflow is the `Aggregator`, for which one or more tasks run at each cluster. The `Aggregator` defines window boundaries in terms of record timestamps, and maintains the dynamically sized cache from Section 5, with each task aggregating a hash-partitioned subset of the key space. We generalize over a broad range of eviction policies by ordering keys using a priority queue with an efficient `changePriority` implementation. 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.

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

2168-7161 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2017.2750678, IEEE Transactions on Cloud Computing

This function can be changed at runtime in order to support arbitrary dynamic sizing policies, such as those described in Sections 5.1.1 and 5.2.1.[8]

The `Aggregator` tasks send their output to a single instance of the `Reorderer` bolt, which is responsible for delaying records as necessary in order to maintain punctuation semantics. Data then flows into the `SocketSender` bolt, which transmits partial aggregates to the center using TCP sockets. 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, though more sophisticated techniques [12] could be employed.

Center: At the center, data follows largely in 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. Staleness is computed relative to the wall-clock (i.e., processing) time at which the window closes. Clocks are synchronized using NTP.

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`[9]. 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.

7.1 Aggregation using a single edge

Our work is motivated by the general case of multiple edges, though 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 begin with a laziness parameter of $\alpha = 0.25$, 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 6(a) and 6(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.[10]

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[11]. As Figures 7(a) and 7(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 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 8(a), we again see that our hybrid algorithm yields near-optimal traffic. We also observe an important effect of stream arrival rate: all else equal, a high 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.

[8] 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.

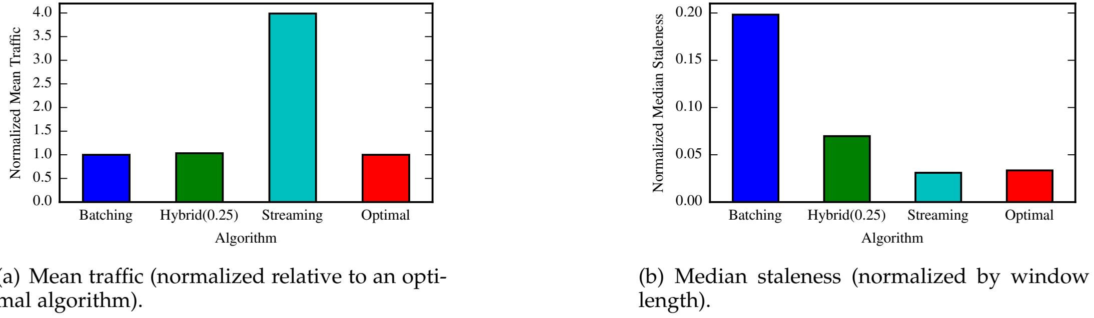
[9] We were forced to confine central aggregation to a single node due to PlanetLab's restrictive daily network bandwidth limit, which was quickly exhausted by communication between Storm workers when using multiple center nodes.

[10] Experimental artifacts introduce variation, but staleness for streaming and optimal are comparable.

[11] 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.

2168-7161 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

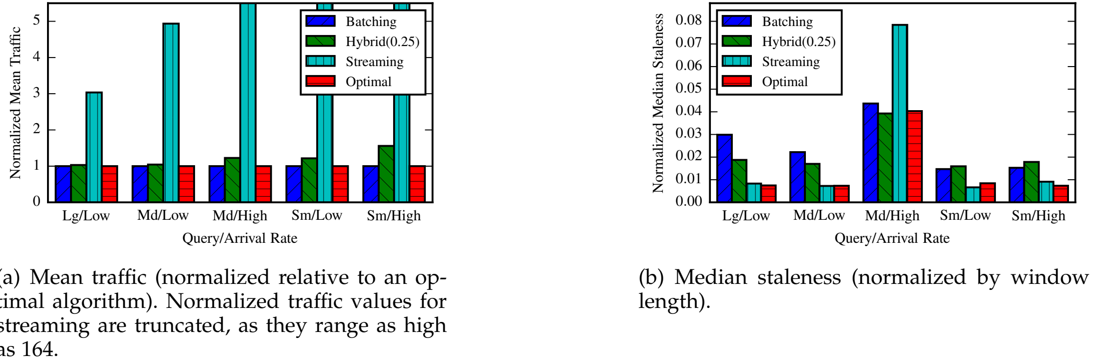
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2017.2750678, IEEE Transactions on Cloud Computing



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

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

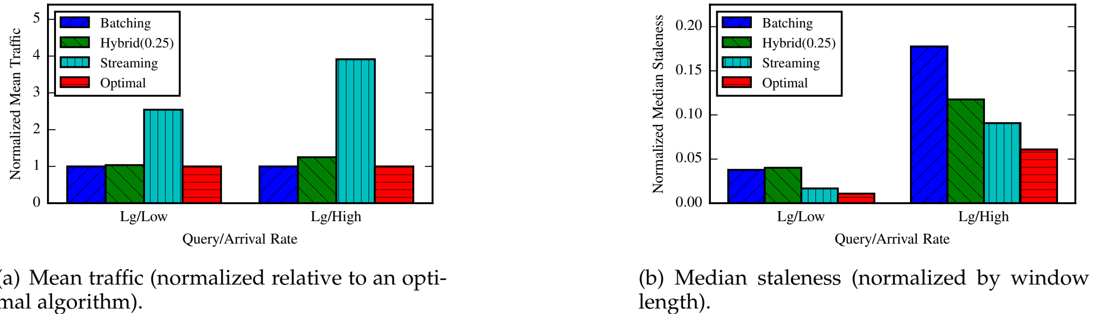
Fig. 6. Performance for batching, streaming, optimal, and our hybrid algorithm for the `large` query with a low stream arrival rate using a one-edge Apache Storm deployment on PlanetLab.



(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).

Fig. 7. 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.



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

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

Fig. 8. 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.

In terms of staleness, Figure 8(b) shows that our algorithm performs well when the arrival rate is high and the network capacity is relatively constrained. In this case, staleness is more sensitive to the particular scheduling algorithm. When the arrival rate is low, we see that our hybrid algorithm performs slightly worse than batching, though in absolute terms the 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. 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. Figure 9(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: 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 9(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

2168-7161 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2017.2750678, IEEE Transactions on Cloud Computing

staleness alone, a more eager algorithm is better. Based on the shape of these trends, $\alpha = 0.25$ appears to be a good compromise value for our experiments.

Selecting the best value of α is an important problem for future work. One promising approach is to use a hill-climbing algorithm that tries different values of α in order to find the best for the given input stream and query. The best value of α also depends on the *relative importance* of traffic and staleness. For example, a small value for α can provide lower staleness, while a larger value for α can yield lower traffic.

7.4 Impact of network capacity

For our final set of results, we use simulations to fully understand the impact of network capacity on our algorithms. We use a simulation methodology for these results since PlanetLab gives us limited control over varying bandwidth capacities, and also has a maximum daily bandwidth cap. Here, we 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 for the `large` query.

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 slightly worse than 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 is nearly staleness-optimal at high network capacity, yet performs worse than all other alternatives under low network capacity. This is because under a highly constrained network, excessive traffic from streaming leads to large—even unbounded—network queuing delays. Batching, on the other hand, is the worst alternative at high network capacity, yet yields near-optimal staleness at low network capacity. 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.

8 Extension to Hopping Windows

While we have focused so far on grouped aggregation for tumbling windows, i.e., non-overlapping windows of length W, a different type of window that is also commonly used is the *hopping window*. A (W, H) hopping window is a time interval of length W that advances every H time steps, where W is a multiple of H. As an example, we might use a (W, H) hopping window to compute an aggregate over 1-hour windows, beginning a new window every 10 minutes (i.e., W=1 hour and H=10 mins).

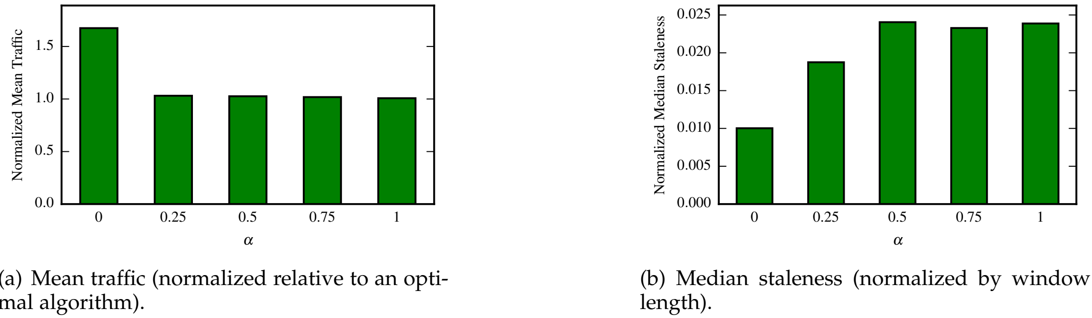
In a (W,H) hopping window, each window $[T, T + W)$ can be partitioned into W/H subwindows of length H, with each subwindow starting at time $T + iH$, $0 \leq i < W/H$. Performing grouped aggregation for a (W, H) hopping window is equivalent to that of computing aggregates over these length-H subwindows at the edge and relying on the center to logically combine the results from these subwindows to form the results for the full length-W windows. Concretely, we define an aggregation algorithm HA for hopping windows as follows. HA runs a tumbling window-based aggregation algorithm A at the edge for each tumbling subwindow sw of length H, sending updates to the center based on the arrivals within that subwindow. The center then applies each incoming update to all length-W windows containing sw. It can be seen that HA generates the aggregates for each length-W window w correctly. Intuitively, this is because (i) no update combines arrivals spanning multiple subwindows (so there are no duplicates), and (ii) the center aggregates over all updates received for the subwindows that constitute w (so it accounts for all arrivals within w).

Given that aggregates have to be generated for *all* windows, we can show that a hopping window algorithm HA that uses one of our optimal algorithms (Section 4) as the aggregation algorithm A for each length-H subwindow can achieve optimal total traffic (across all windows) as well as optimal staleness (for each window). Intuitively, this is because of the following reasons. First, each subwindow sw is the *last* subwindow of some window w. Therefore, any updates for sw must be sent to the center to correctly compute the aggregates for w. In other words, updates have to be sent from *every* subwindow to the center. Since our optimal algorithms minimize the traffic for any sw, they also minimize the total traffic sent over all subwindows (and hence across all windows). Second, the staleness for window w is the same as the staleness for its last subwindow sw'. Thus, our optimal algorithms achieve the optimal staleness for window w given that they optimize the staleness for each subwindow sw'.

Figure 11 shows traffic and staleness for a hopping window aggregation algorithm HA using streaming, batching, and our optimal algorithms as W remains fixed and H decreases, yielding a higher W/H ratio. In this figure, we use the `large` query and simulate a medium-bandwidth WAN (corresponding to a normalized network capacity of roughly 2.1). These results illustrate several key points. First, the optimal algorithm achieves the lowest staleness and traffic in all cases. Further, as W/H increases, optimal traffic increases. The reason is that, as H decreases, aggregation has to be performed over shorter subwindows, and is therefore less effective in reducing the number of flushed updates relative to the number of arrivals. Optimal staleness also increases with W/H due to the increasing traffic. When W/H is sufficiently high, traffic exceeds the network capacity, leading to very high staleness due to network congestion. Overall, we see that as W/H increases, the effects of a constrained WAN become more pronounced, and jointly minimizing traffic and staleness becomes increasingly challenging. In the limit, the windows become continuously *sliding windows*, where no aggregation is feasible and streaming is the only feasible algorithm.

2168-7161 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

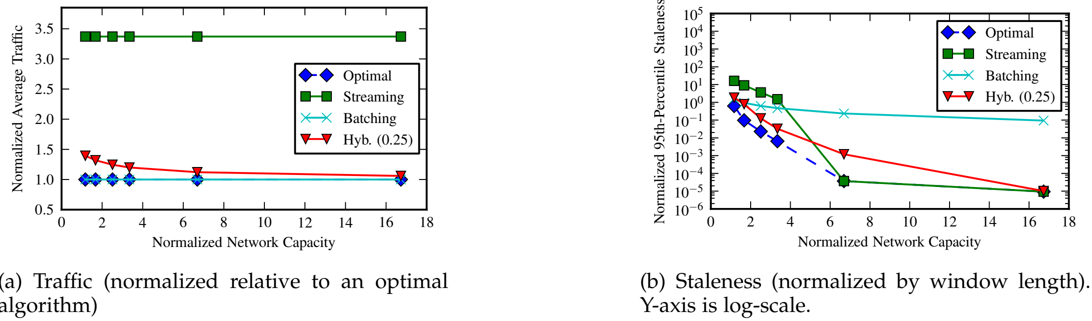
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2017.2750678, IEEE Transactions on Cloud Computing



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

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

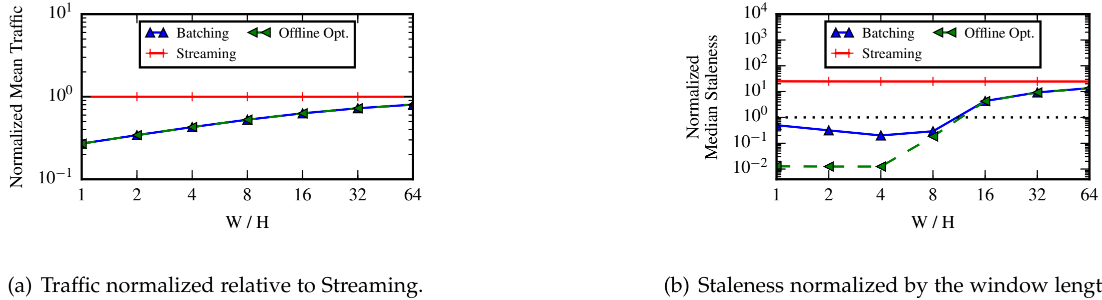
Fig. 9. Effect of laziness parameter α using a three-edge Apache Storm deployment on PlanetLab with query `large`.



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

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

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



(a) Traffic normalized relative to Streaming.

(b) Staleness normalized by the window length.

Fig. 11. Traffic and staleness for a hopping window-based grouped aggregation.

9 Discussion

Compression: Compression could be applied to batches of key-value pairs to reduce WAN traffic. The benefit of compression depends on the speed of compression and decompression relative to the WAN bandwidth, as well as the extent to which compression, transmission, and decompression can be pipelined. Further, as we increase the size of the batches to which we apply compression, the compression ratio increases, but so does the additional batching delay, which may lead to higher rather than lower staleness. We apply Google's Snappy compression [13] to our anonymized Akamai trace at several granularities, and show the results in Figure 12. We run each experiment five times and show the mean, with error bars indicating 95% confidence intervals for latency (compression ratio does not vary from run to run). We see that it is possible to achieve a high compression ratio over batches small enough to be compressed with sub-millisecond latency. Incorporating compression into our techniques is therefore quite promising, though it is complementary to our algorithms, and implementing it in our experimental prototype remains an item for future work.

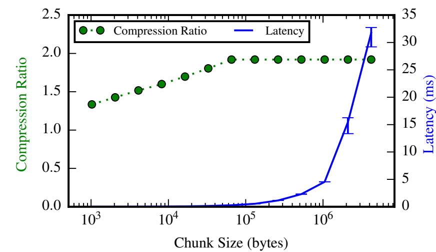


Fig. 12. Compression ratio and latency for Google Snappy compression applied at several granularities to our anonymized Akamai trace.

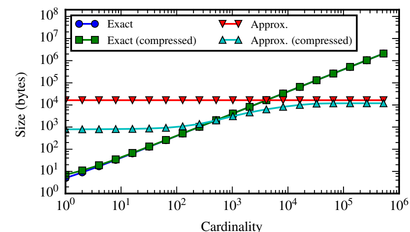


Fig. 13. Size of exact and approximate set aggregations with and without compression.

2168-7161 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2017.2750678, IEEE Transactions on Cloud Computing

Variable-size aggregates: We have assumed that aggregate values have constant size. In practice, this is true for many aggregates such as `Sum`, and is reasonable even for many sophisticated aggregates, though there are opportunities for optimization. For example, consider an approximate set cardinality counter implemented using a `HyperLogLog` sketch [14]. The size of a `HyperLogLog` instance is dictated not by the set that it approximates, but by the error-tolerance parameter specified at its construction. Figure 13 shows the space required to represent a set of random ip addresses using both exact and approximate representations with and without (Google Snappy) compression. Across a range of cardinalities, the approximate representation requires nearly constant space. For large sets, the approximate representation is much smaller than an exact representation, but for small sets, the exact representation may be more concise, even smaller than a compressed approximate representation. In practice, choosing the best representation based on the given value is a powerful optimization.

This optimization is fundamentally compatible with our algorithms. We expect practical implementations to choose the most concise representation for a given aggregation (e.g., an exact set at low cardinalities, and a `HyperLogLog` at high cardinalities). We then view our dynamic cache sizing policy as reflecting not the number, but rather the total size, of aggregates to allow in the cache. The eviction policy remains responsible for identifying which key is most likely to have attained its final aggregate value, a decision that is independent of the representation of the value for that key. This optimization allows more keys to remain cached at the edge at any given time, reducing the pressure to evict early, thus leading to lower traffic and staleness.

Applicability to other environments: While our focus is on geo-distributed stream analytics, our algorithms are applicable to other environments such as single data center or cluster environments. Extending our techniques to deployments with multiple central sites, or in hierarchical topologies, is an interesting area of future work. For a setting with multiple central sites, our algorithms may be applicable if the output is sharded across multiple centers, though one must consider the bandwidth constraints for transmitting data to multiple centers. Similarly, these algorithms could be applied to heirarchical topologies, though one must consider more bursty arrivals and need for coordination at intermediate levels of the hierarchy.

10 Related Work

Aggregation: Aggregation is a key operator in analytics, and grouped aggregation is supported by many data-parallel programming models [2], [15], [16]. Larson et al. [17] 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 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 [18], [19], whereas we focus on cost and timeliness. Amur et al. [20] study grouped aggregation, discussing tradeoffs between eager and lazy aggregation, but do not consider the effect on staleness, a key performance metric in our work. A preliminary version of this paper [21] presented the main algorithms in this work, which we have extended here with additional results, insights, and discussion.

Streaming systems: Numerous streaming systems [3], [11], [22], [23], [24] 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 timely results at low cost.

Wide-area and Edge computing: Recent research on wide-area computing has primarily focused on batch computing [7], [25], [26], [27]. Relatively little work on streaming computation [28] has focused on wide-area deployments or scheduling. Pietzuch et al. [29] optimize operator placement in geo-distributed settings to balance between system-level bandwidth usage and latency. Hwang et al. [30] rely on replication across the wide area to achieve fault tolerance and reduce straggler effects. JetStream [4] considers wide-area streaming computation, but unlike our work, assumes it is always better to push more computation to the edge. Edge computing has been used in other contexts with a different focus. Examples include Cloud4Home [31] for edge storage, Nebula [32] for volunteer-based computing, and Cloudlets [33] for latency-sensitive mobile offloading.

Optimization tradeoffs: LazyBase [34] provides a mechanism to trade off increased staleness for faster query response in the case of ad-hoc queries. BlinkDB [35] and JetStream [4] provide mechanisms to trade off accuracy with response time and bandwidth utilization, respectively. We focus on *jointly* optimizing both network traffic and staleness. Das et al. [36] consider tradeoffs between throughput and latency in Spark Streaming, but do not consider scheduling on a per-key basis, like we do.

Our algorithms can be extended to *approximate* computation, where the tradeoff is between accuracy and staleness (or traffic). We explore this tradeoff fully in other recent work [37].

11 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 PlanetLab. 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 queries.

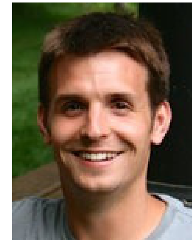
2168-7161 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

Acknowledgments

We would like to thank Ravali Kandur for her help deploying Apache Storm on PlanetLab, and NSF grants CNS-1413998 and CNS-1619254, as well as an IBM Faculty Award, which partially supported this research.

References

[1] "Storm, distributed and fault-tolerant realtime computation," http://storm.apache.org/, 2015.
[2] O. Boykin, S. Ritchie, I. O'Connel, and J. Lin, "Summingbird: A framework for integrating batch and online mapreduce computations," in *Proc. of VLDB*, vol. 7, no. 13, 2014, pp. 1441–1451.
[3] 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*, 2013, pp. 423–438.
[4] 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*, 2014, pp. 275–288.
[5] M. Adler, R. K. Sitaraman, and H. Venkataramani, "Algorithms for optimizing the bandwidth cost of content delivery," *Computer Networks*, vol. 55, no. 18, pp. 4007–4020, Dec. 2011.
[6] 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.*, vol. 39, no. 1, pp. 68–73, Dec. 2008.
[7] A. Vulimiri, C. Curino, B. Godfrey, K. Karanasos, and G. Varghese, "WANalytics: Analytics for a geo-distributed data-intensive world," in *Proc. of CIDR*, 2015.
[8] "PlanetLab," http://planet-lab.org/, 2015.
[9] E. Nygren, R. K. Sitaraman, and J. Sun, "The akamai network: a platform for high-performance internet applications," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 3, pp. 2–19, Aug. 2010.
[10] S. Podlipnig and L. Böszörmenyi, "A survey of web cache replacement strategies," *ACM Comput. Surv.*, vol. 35, no. 4, pp. 374–398, Dec. 2003.
[11] T. Akidau *et al.*, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proc. of the VLDB Endowment*, vol. 8, pp. 1792–1803, 2015.
[12] J. Bolliger and T. Gross, "Bandwidth monitoring for network-aware applications," in *Proc. of HPDC*, 2001, pp. 241–251.
[13] "Google/snappy: A fast compressor/decompressor," https://github.com/google/snappy/, 2016.
[14] P. Flajolet, É. Fusy, O. Gandouet *et al.*, "HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm," in *Proc. of AOFA*, 2007.
[15] J. Gray *et al.*, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Data Min. Knowl. Discov.*, vol. 1, no. 1, pp. 29–53, Jan. 1997.
[16] Y. Yu, P. K. Gunda, and M. Isard, "Distributed aggregation for data-parallel computing: interfaces and implementations," in *Proc. of SOSP*, 2009, pp. 247–260.
[17] P.-A. Larson, "Data reduction by partial preaggregation," in *Proc. of ICDE*, 2002, pp. 706–715.
[18] 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, pp. 131–146.
[19] R. Rajagopalan and P. Varshney, "Data-aggregation techniques in sensor networks: A survey," *IEEE Communications Surveys Tutorials*, vol. 8, no. 4, pp. 48–63, 2006.
[20] H. Amur *et al.*, "Memory-efficient groupby-aggregate using compressed buffer trees," in *Proc. of SoCC*, 2013.
[21] B. Heintz, A. Chandra, and R. K. Sitaraman, "Optimizing grouped aggregation in geo-distributed streaming analytics," in *Proc. of HPDC*, 2015, pp. 133–144.
[22] T. Akidau *et al.*, "MillWheel: Fault-tolerant stream processing at internet scale," *Proc. of VLDB Endow.*, vol. 6, no. 11, pp. 1033–1044, Aug. 2013.
[23] "Apache Flink: Scalable Batch and Stream Data Processing," http://flink.apache.org/, 2016.
[24] S. Kulkarni *et al.*, "Twitter heron: Stream processing at scale," in *Proc. of SIGMOD*, 2015, pp. 239–250.
[25] B. Heintz, A. Chandra, R. K. Sitaraman, and J. Weissman, "End-to-end optimization for geo-distributed mapreduce," *IEEE TCC*, 2015.
[26] 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*, 2015, pp. 421–434.
[27] A. C. Zhou, S. Ibrahim, and B. He, "On achieving efficient data transfer for graph processing in geo-distributed datacenters," in *Proc. of ICDCS*, June 2017, pp. 1397–1407.
[28] D. J. Abadi *et al.*, "The design of the borealis stream processing engine," in *Proc. of CIDR*, 2005, pp. 277–289.
[29] P. Pietzuch *et al.*, "Network-aware operator placement for stream-processing systems," in *Proc. of ICDE*, 2006.
[30] J.-H. Hwang, U. Cetintemel, and S. Zdonik, "Fast and highly-available stream processing over wide area networks," in *Proc. of ICDE*, 2008, pp. 804–813.
[31] S. Kannan, A. Gavrilovska, and K. Schwan, "Cloud4home–enhancing data services with@ home clouds," in *Proc. of ICDCS*. IEEE, 2011, pp. 539–548.
[32] M. Ryden, K. Oh, A. Chandra, and J. Weissman, "Nebula: Distributed edge cloud for data intensive computing," in *Proc. of IC2E*. IEEE, 2014, pp. 57–66.
[33] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The Case for VM-Based Cloudlets in Mobile Computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, October 2009.
[34] J. Cipar *et al.*, "LazyBase: trading freshness for performance in a scalable database," in *Proc. of EuroSys*, 2012, pp. 169–182.
[35] S. Agarwal *et al.*, "BlinkDB: queries with bounded errors and bounded response times on very large data," in *Proc. of EuroSys*, 2013, pp. 29–42.
[36] T. Das, Y. Zhong, I. Stoica, and S. Shenker, "Adaptive stream processing using dynamic batch sizing," in *Proc. of SoCC*, 2014, pp. 16:1–16:13.
[37] B. Heintz, A. Chandra, and R. K. Sitaraman, "Trading timeliness and accuracy in geo-distributed streaming analytics," in *Proc. of ACM SOCC*, Oct. 2016.



Benjamin Heintz is currently a Software Engineer at Facebook, where he works on stream processing systems. Heintz holds M.S. and Ph.D. degrees in Computer Science from the University of Minnesota, and a B.S. in Mechanical Engineering and Economics from Northwestern University.

Abhishek Chandra is an Associate Professor in the Department of Computer Science and Engineering at the University of Minnesota. His research interests are in the areas of Operating Systems and Distributed Systems. He received his B.Tech. degree in Computer Science and Engineering from Indian Institute of Technology Kanpur, and his M.S. and Ph.D. degrees in Computer Science from the University of Massachusetts Amherst.

Ramesh K. Sitaraman is a professor of computer science at University of Massachusetts Amherst and chief consulting scientist at Akamai Technologies. His research spans all aspects of Internet-scale distributed systems, including algorithms, architectures, security, performance, energy efficiency, user behavior, and economics. Sitaraman received a PhD in computer science from Princeton University and a B.Tech in Electrical Engineering from the Indian Institute of Technology Madras.

2168-7161 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.