A TTL-based Approach for Data Aggregation in Geo-distributed Streaming Analytics

DHRUV KUMAR, University of Minnesota, Twin Cities
JIAN LI, University of Massachusetts, Amherst
ABHISHEK CHANDRA, University of Minnesota, Twin Cities
RAMESH K. SITARAMAN, University of Massachusetts, Amherst

Streaming analytics require real-time aggregation and processing of geographically distributed data streams continuously over time. The typical analytics infrastructure for processing such streams follow a hub-and-spoke model, comprising multiple edges connected to a center by a wide-area network (WAN). The aggregation of such streams often require that the results be available at the center within a certain acceptable delay bound. Further, the WAN bandwidth available between the edges and the center is often scarce or expensive, requiring that the traffic between the edges and the center be minimized.

We propose a novel Time-to-Live (TTL-)based mechanism for real-time aggregation that provably optimizes both delay and traffic, providing a theoretical basis for understanding the delay-traffic tradeoff that is fundamental to streaming analytics. Our TTL-based optimization model provides analytical answers to how much aggregation should be performed at the edge versus the center, how much delay can be incurred at the edges, and how the edge-to-center bandwidth must be apportioned across applications with different delay requirements.

To evaluate our approach, we implement our TTL-based aggregation mechanism in Apache Flink, a popular stream analytics framework. We deploy our Flink implementation in a hub-and-spoke architecture on geo-distributed Amazon EC2 data centers and a WAN-emulated local testbed, and run aggregation tasks for realistic workloads derived from extensive Akamai and Twitter traces. The delay-traffic tradeoff achieved by our Flink implementation agrees closely with theoretical predictions of our model. We show that by deriving the optimal TTLs using our model, our system can achieve a "sweet spot" where both delay and traffic are minimized, in comparison to traditional aggregation schemes such as batching and streaming.

CCS Concepts: • **Computer systems organization → Cloud computing**; • **Information systems → Data analytics**; **Data streaming**;

Keywords: Geo-distributed systems; Edge; Cloud; Stream processing.

ACM Reference Format:
Dhruv Kumar, Jian Li, Abhishek Chandra, and Ramesh K. Sitaraman. 2019. A TTL-based Approach for Data Aggregation in Geo-distributed Streaming Analytics. In *Proc. ACM Meas. Anal. Comput. Syst.*, Vol. 3, 2, Article 29 (June 2019). ACM, New York, NY. 27 pages. https://doi.org/10.1145/3326144

1 INTRODUCTION

Streaming data analytics has been an important topic of research in recent years. Large quantities of data are generated continuously over time across a variety of application domains such as web

Authors' addresses: Dhruv Kumar, dhruv@umn.edu, University of Minnesota, Twin Cities; Jian Li, jianli@cs.umass.edu, University of Massachusetts, Amherst; Abhishek Chandra, chandra@umn.edu, University of Minnesota, Twin Cities; Ramesh K. Sitaraman, ramesh@cs.umass.edu, University of Massachusetts, Amherst.

2476-1249/2019/6-ART29 $15.00
https://doi.org/10.1145/3326144

and social analytics, scientific computing and energy analytics. One of the key requirements in modern data analytics services is the real-time analysis of these data streams to extract useful and timely information for the analyst. Several distributed data analytics platforms [7, 36] have been developed in recent times to meet this growing requirement of real-time streaming analytics. Nowadays, a large amount of data is generated continuously by geographically distributed sources (e.g., agents, sensors, mobile devices, edge nodes, etc.) in many streaming applications [34]. For instance, services like Facebook, Twitter and Netflix continuously gather data from the end users for a variety of analytical purposes such as finding the popular web content amongst their users or monitoring the QoS metrics. Large content delivery networks (CDNs) like Akamai [25] that serve a significant fraction of content on the Internet continuously collect data from their edge servers and clients from around the globe to understand what, where and how content is accessed for the purpose of providing content analytics insights to businesses.

Hub-and-spoke model for analytics processing. A typical analytics infrastructure for processing such geo-distributed streams follows a hub-and-spoke model, which conceptually comprises a single centralized "hub" connected to multiple edges by a wide-area network (WAN). The data is either generated at the edge or collected at the edge from clients such as sensors, mobile devices etc. In the latter case, clients report data to the edge that is "closest" to them. Each edge has a cluster of servers to collect and process its data streams and then send the processed data to a central hub for further processing. Analysts directly query the central server for retrieving the relevant analyzed data. In this paper, we limit ourselves to aggregation-based processing which we explain next.

Data aggregation. We focus on optimizing a common and widely-used operation that is performed within any analytics system – the operation of computing aggregates, such as the Reduce operation in MapReduce, GroupBy in SQL and LINQ etc. We consider data streams in which every record is of the type (k, v), where k is the key and v is its corresponding value, e.g., in the Akamai content analytics context the key could be a combination of content id (url) and geographic location and the value could be the number of clients accessing the url from that location. Aggregation is performed over such a key-value stream by grouping all records (k, v_i), $1 \leq i \leq n$ that have the same key value k, to produce an aggregate record $(k, v_1 \oplus v_2 \oplus \cdots \oplus v_n)$, where $v_1, v_2, \cdots, v_n$ are the values received for key k up to time T and $\oplus$ is an application-defined associative binary operator. Such operators can be as simple as *sum* or *max*, or more sophisticated, including filters (such as Bloom filters), transforms and sketches (such as HyperLogLog), and user-defined functions. In this paper, we focus on continuous aggregation at the center where the newly arrived data record (k, v) is immediately aggregated into its key k's aggregated value to provide the most updated aggregated result for any key k.

Delay-traffic tradeoff. The data transfer from the edges to the center happens over a WAN link which is generally scarce or expensive [27]. To save WAN bandwidth, the computing resources on the edge could be utilized to perform (partial) aggregation on the input data stream before sending intermediate results to the center for a full aggregation. Such edge-based aggregation leads to a fundamental tradeoff between two key metrics: *delay* and *WAN traffic*, as illustrated in Figure 1. Here, delay corresponds to the edge-induced aggregation delay[1] in computing the results, while WAN traffic is the amount of data sent out over the wide-area links. Figure 1 shows the delay-traffic tradeoff, with delay decreasing as traffic increases and vice versa. In particular, the two end-points correspond to two extreme approaches to edge aggregation: *streaming* and *batching*. Streaming refers to sending all the data from the edges to the center without any processing at the edges. This approach is able to achieve the desired low delay but results in high WAN traffic. On the other

[1]While network delay is another component of the end-to-end delay, we mainly focus on aggregation delay as this is a direct consequence of edge aggregation, and consider network delay in Section 6.

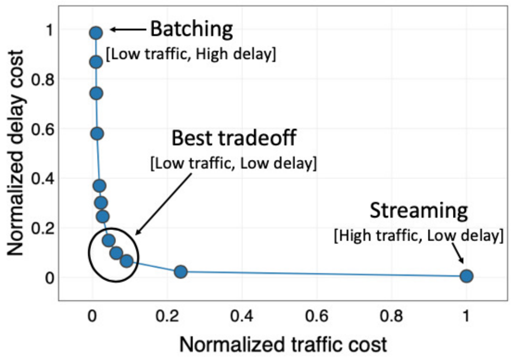


Fig. 1. *Delay-traffic tradeoff.* This figure shows the empirically computed delay and traffic for varying amounts of edge-based aggregation for the Akamai trace using AWS EC2 testbed. More details in Section 5.

hand, batching[2] refers to aggregating the data at the edge for a long time (typically, a few hours) and then sending the summarized data to the center for further processing. This approach is able to achieve the desired low traffic but results in high delay. However, different analytics systems, or even different applications using the same system, require different tradeoffs between delay and traffic.

Real-world examples. We discuss examples of analytics services from industry to further support the different delay-traffic tradeoff requirements explained above.

1) Akamai Media Analytics. Akamai Media Analytics [3] provides insights into online video performance, quality of experience, and audience behavior by monitoring crucial metrics that drive business critical decisions. Media Analytics consists of two main services: a delay-sensitive Quality of Service (QoS) Monitor and a delay-tolerant Audience Analytics. The customers of both the services are video providers who use Akamai's video delivery services. Both analytics services use a hub-and-spoke model where individual clients (video players) send information as a stream of packets (called "beacons") in real-time to the widely-distributed Akamai edge servers. Each beacon has key-value pairs that are aggregated by the edge servers and then forwarded to a central hub for more aggregation and processing. Users visualize the fully aggregated data by accessing the central hub. Our experiments use the beacon traces from Akamai that underly both these services.

QoS Monitor analyzes the quality of video streams using metrics such as startup time, rebuffer rates, audience size, bitrates, and availability in near real-time. The goal is for video providers using the service to get a real-time view of how their end-users are experiencing their video streams and take immediate diagnostic action if there is a noticeable quality degradation. A common use case is when a large live streaming event such as the FIFA soccer world cup is being delivered by Akamai. The analytics service is used to quickly identify and solve video quality degradations, even as the event is in progress. Given the performance diagnostic goals of the service, reducing delay is extremely important, even at a greater traffic cost.

Audience Analytics provides analytics around the behavior of the audience engaging with video content using metrics such as time spent per video, geographic location of the user and so on. This is not a service used for performance diagnostics, but rather for the video provider to gain analytic insights useful for the business. So, it is more crucial to reduce the traffic cost, even at the expense of greater delay.

[2]Strictly speaking, anything that is not pure streaming can be called as batching but in this paper, we user the term batching for referring to aggregation for a long time (such as few hours).

2) Twitter Analytics. A large number of businesses use Twitter for their marketing and advertisement campaigns [30]. In such cases, the businesses want to understand how their audience engages with their brands and campaigns. Twitter Analytics such as trending hashtags, trending topics, audience statistics etc are very common for making business decisions. The need for real-time updates versus reducing traffic cost may vary depending on the specific use. For instance, real-time advertisement campaigns are delay sensitive. But, brand awareness campaigns have longer-term goals and less delay sensitive, making traffic cost reduction important. Consequently, our model and algorithms allow for adjusting the relative weights of delay and traffic at a granular level, even allowing per-key weights. Note that we use traces from Twitter for a trending topics query in our evaluation.

In summary, between the extremes of streaming and batching, there are different operating points for a wide-area data analytics system that represent different tradeoffs between delay and traffic. *A goal of our work is to devise edge aggregation mechanisms that can provably achieve the desired delay-traffic tradeoffs.*

Research contributions. In this paper, we propose a novel *Time-to-Live (TTL)-based aggregation* mechanism for online stream aggregation that provably optimizes delay and traffic jointly. In this approach, records corresponding to each key are aggregated at the edge for a certain time period dictated by its TTL, before the aggregates are sent over the WAN to the center. The proposed approach is able to achieve the desired delay-traffic tradeoff, and is also able to satisfy the *low delay - low traffic* requirement where needed. To the best of our knowledge, we are the first to provide a theoretical basis for understanding the delay-traffic tradeoff that is fundamental to streaming analytics. In doing so, we provide analytical answers to how much aggregation should be performed at the edge versus the center, how much delay can be incurred at the edges, and how the edge-to-center bandwidth must be apportioned across different applications with different delay requirements. We study the tradeoff between delay and traffic by presenting a family of optimal TTL-based algorithms for jointly minimizing both delay and traffic. In addition, we present extensions to our approach to solve two complementary problems: (i) minimizing delay under a traffic constraint and (ii) minimizing traffic under a delay constraint. This paper makes the following research contributions:

- To the best of our knowledge, the proposed TTL-based aggregation model is the first to provide a theoretical basis for understanding the delay-traffic tradeoff that is fundamental to geo-distributed streaming analytics. Our model also characterizes the storage requirements at the edges to achieve such a tradeoff.
- Using this model, we show how to optimize delay and traffic jointly, achieving a user-desired delay-traffic tradeoff, while also characterizing a stable "sweet spot" operating region that achieves the "best" tradeoff where both delay and traffic are relatively small.
- We have implemented the TTL-based aggregation mechanism in Apache Flink, a popular stream analytics framework. As part of this implementation, we provide a simple, expressible API for users to easily leverage the proposed optimization framework. In addition, our optimization dynamically adapts to changing workloads.
- We evaluate our approach through experiments running our Flink implementation on geo-distributed Amazon EC2 data centers, as well as a local cluster emulating WAN characteristics. The experiments are driven by real-world Akamai and Twitter traces. Our empirical results are in close agreement with our theoretical model predictions. Further, we show that by deriving the optimal TTLs using our model, our system can achieve a "sweet spot" stable operating point where both delay and traffic are minimized, in comparison to traditional aggregation schemes such as batching and streaming.

- We show that the proposed TTL-based aggregation is general enough to model a wide variety of optimization formulations with different types of objective functions and constraints.

2 TIME-TO-LIVE (TTL) AGGREGATORS

The main aggregation mechanism that we propose is a *TTL aggregator* that takes a key-value data stream as input and outputs an aggregated key-value stream. The aggregator has a cache to store keys for aggregation. Each key k has a TTL value T_k that is the time period for which the key k must be kept in the cache. Figure 2 shows the flow of records in a TTL aggregator. When a record (k, v) arrives at the edge for which the key k is not present in the cache, a *cache miss* is said to have occurred. In case of a cache miss, the key k is inserted into the cache and its aggregated value is set to the value of the record, i.e., $agg_k = v$. Additionally, a timer is created and assigned to key k with its value set to the TTL value T_k of the key, i.e., $(Timer_k = T_k)$. The timer counts down and until the timer expires (i.e., $Timer_k = 0$), any new records of key k are aggregated into the result stored in the cache for key k without resetting the timer value. That is, for any new record (k, v') for which the key k is present in the cache, a *cache hit* is said to have occurred and $agg_k = agg_k \oplus v'$. When timer $Timer_k$ goes to zero, the aggregated record (k, agg_k) is sent out in the output stream.

Relationship to TTL caches. There is extensive literature [6, 10, 12, 14, 15, 22] on TTL caches that store frequently used objects in context of content, database records, memory pages, etc. Such a cache would set a TTL when an object is first stored in cache and evict the object when the TTL expires. While a TTL aggregator serves a different purpose, some of the theoretical analysis of TTL caches directly apply. This novel connection between aggregation and traditional caching allows us to bring to bear the analytical work done in the caching domain into data analytics.

Comparison with windowed grouped aggregation. Windowed grouped aggregation is a common operation used in many stream processing frameworks, where records within a time window are aggregated together. In some batch computing-based execution frameworks (e.g., Spark Streaming [36]), windowed grouped aggregation is used for microbatching streams of records to simulate stream computing. TTL-based aggregation approach provides a number of advantages over windowed grouped aggregation.

First, the window size in windowed grouped aggregation is typically application-defined (and fixed statically), whereas TTL-based aggregation supports dynamic TTL window sizes that can help us achieve desired tradeoffs. Most systems also use the same window size across all the keys in windowed grouped aggregation, whereas the TTL-based approach allows different TTL window sizes for different keys. Even in cases where the window size can be tuned, determining the optimal or "best" window size is hard in practice and is typically done using ad hoc approaches. TTL-based aggregation, on the other hand, provides a principled approach to *automatically* and *adaptively* determine per-key window sizes. It can be employed to achieve theoretically provable optimal TTL window sizes that satisfy the desired delay-traffic tradeoffs (see Section 3). More broadly, the TTL-based aggregation approach opens up the door for finding optimal solutions to a wide variety of problem formulations with different types of constraints on delay and traffic. We give a flavor of these in Section 6. No such solutions exist for windowed grouped aggregation to the best of our knowledge.

Delay versus traffic tradeoff using a TTL aggregator. Suppose that each key k arrives in a Poisson process[3] at the input of a TTL aggregator with rate λ_k (in records per unit time) and a TTL value of T_k is used for that key. Let the expected aggregation delay, D_k, be the expected delay experienced by the arrival of a record with key k, i.e., D_k is the expected time difference between

[3]We relax the Poisson arrivals assumption for our implementation and empirical evaluation.

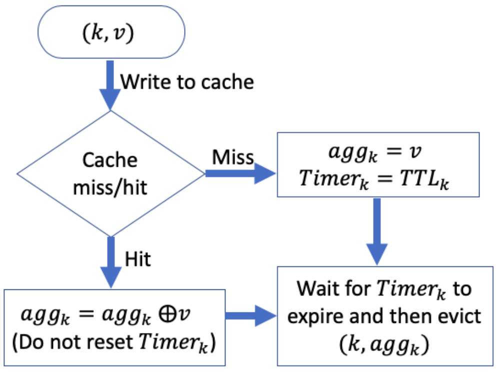


Fig. 2. TTL aggregator.

the arrival of record with key k and the next departure of the record with that key. Let μ_k be the departure rate[4] of key k (in records per unit time) and m_k be the miss probability of key k.

Theorem 2.1. *For all keys k, $m_k = \mu_k = \frac{1}{1+\lambda_k T_k}$ and the expected aggregation delay $D_k = (m_k + 1)T_k/2 = (\frac{1}{1+\lambda_k T_k} + 1)T_k/2$.*

Proof. The probability of not finding a key k in cache of the TTL aggregator is the miss probability m_k. Since the miss probability analysis derived in the context of TTL caches [14] equally applies in our TTL aggregator context, we derive $m_k = \frac{1}{1+\lambda_k T_k}$. Since every flush of key k into the output stream can be matched with a prior time when the key entered the cache on a "cache miss" event, μ_k equals the miss probability m_k.

The expected aggregation delay D_k, for each arrival of key k, can be computed as follows. When a key k arrives at a time t and it results in a miss, that key is inserted into the cache, a timer is set of the value of T_k, and the key is flushed from cache at time $t + T_k$. Thus, if the arrival of key k at time t is a cache miss, its delay is exactly T_k. Note that a cache hit occurs for every arrival of key k within the time interval $[t, t + T_k]$, since the key is present in cache during that time period. Since the arrivals for key k are Poisson, the expected delay of the cache hits that arrive in the interval $[t, t + T_k]$ is exactly half the size of that interval, i.e., the delay is $T_k/2$. To derive this more formally, the cumulative delay experienced by all cache hit arrivals in the interval $[t, t + T_k]$ is $\int_t^{t+T_k} \lambda_k(t + T_k - x)dx = \lambda_k T_k^2/2$. Since the expected number of cache hit arrivals in interval $[t, t + T_k]$ is $\lambda_k T_k$, the expected aggregation delay per cache hit arrival is $(\lambda_k T_k^2/2)/\lambda_k T_k$, which equals $T_k/2$. Weighting the delay for cache misses and hits by their respective probabilities, the aggregation delay $D_k = m_k T_k + (1 - m_k)T_k/2 = (m_k + 1)T_k/2$. □

Evaluating the storage at a TTL aggregator. Let B the expected number of keys in the aggregator at a given time. The time-average probability (occupancy probability) that key k is in the aggregator equals to $1 - m_k$. Therefore, we have

$$B = \sum_k (1 - m_k) = \sum_k \frac{\lambda_k T_k}{1 + \lambda_k T_k}. \tag{1}$$

3 OPTIMIZING DELAY-TRAFFIC TRADEOFF

The main advantage of TTL aggregators is that they can be deployed at the edges of a hub-and-spoke analytics system to provide precise, theoretically-validated delay-traffic tradeoffs. Let there be M

[4]The departures may no longer be Poisson.

edges with the m^{th} edge having N_m distinct keys. Denote the sets of edges and distinct keys as $\mathcal{M}$ and $\mathcal{N}_m$, respectively. Suppose that λ_k^m is arrival rate of k^{th} key at the m^{th} edge, where $1 \le k \le N_m$ and $1 \le m \le M$. An analytics system serves multiple applications that have different delay and traffic requirements. A real-time application is more delay-sensitive and may place a greater penalty for delay for its keys than a non-real-time one. An application with larger records may be more expensive to transmit from edge to center than an application with records of a smaller size. So, to posit a tradeoff between delay and traffic, we incorporate unit costs for traffic, c_k^m, and for delay, d_m^k, for each key value k at edge m, allowing variation in these costs according to both key and edge.

Our goal is to find the optimal vector of timers $\boldsymbol{T} = \{T_k^m\}_{k\in\mathcal{N}_m, m\in\mathcal{M}}$ to minimize the total cost $C(\boldsymbol{T})$ incurred across all keys and edges as defined below

$$C(\boldsymbol{T}) = \alpha C_{\text{delay}} + (1-\alpha) C_{\text{traffic,}}$$

where the total delay cost[5] is C_{delay}, the total traffic cost is C_{traffic} and $0 \le \alpha \le 1$ is a system-wide weighting factor that determines how the two costs are weighted against each other. Using Theorem 2.1, we can write C_{delay} as the sum of the delay cost of all keys over all edges, i.e.,

$$C_{\text{delay}} = \sum_{m=1}^{M} \sum_{k=1}^{N_m} \frac{d_k^m \lambda_k^m T_k^m}{2} \left(1 + \frac{1}{1+\lambda_k^m T_k^m}\right). \tag{2}$$

Similarly, we can write C_{traffic} as the sum of the traffic cost of all keys over all edges, i.e.,

$$C_{\text{traffic}} = \sum_{m=1}^{M} \sum_{k=1}^{N_m} \frac{c_k^m \lambda_k^m}{1+\lambda_k^m T_k^m}. \tag{3}$$

Therefore, we can write the overall cost $C(\boldsymbol{T})$ to equal the following

$$\alpha \sum_{m=1}^{M} \sum_{k=1}^{N_m} \frac{d_k^m \lambda_k^m T_k^m}{2} \left(1 + \frac{1}{1+\lambda_k^m T_k^m}\right) + (1-\alpha) \sum_{m=1}^{M} \sum_{k=1}^{N_m} \frac{c_k^m \lambda_k^m}{1+\lambda_k^m T_k^m}. \tag{4}$$

The total delay cost C_{delay} represents the loss of revenue for the network provider if it is not able to meet the SLA signed with the application owner whose application uses its network. The total traffic cost C_{traffic} represents the operational cost for the network provider for enabling application data transfer via its network.

Theorem 3.1. *The optimal timers (TTLs) that minimize the expression for $C(\boldsymbol{T})$ in Equation (4) satisfy*

$$T_k^m = \begin{cases} \frac{1}{\lambda_k^m}\left(\sqrt{\frac{2(1-\alpha)c_k^m \lambda_k^m}{\alpha d_k^m} - 1} - 1\right), & \lambda_k^m \ge \frac{\alpha d_k^m}{(1-\alpha)c_k^m}, \\ 0, & \textit{otherwise}, \end{cases} \tag{5}$$

for $k \in \mathcal{N}_m$ and $m \in \mathcal{M}$.

Proof. We first prove that $C(\boldsymbol{T})$ is convex in T_k^m and then derive the optimal value by differentiating it with respect to T_k^m and setting the differential to zero. Details are in Appendix 10. □

Without loss of generality, we assume that the edges are ordered in the decreasing order of arrival rate, i.e., $\lambda_1^m \ge \cdots \ge \lambda_{N_m}^m$. Suppose for each $k = 1, \cdots, N_m$, there exists a k', such that

[5]Note that the network delay, measured as the time between the key being flushed from the edge server to the time that the center receives it, is independent of our optimizing parameter T_k. Network delay can be easily incorporated within our framework without impacting our analysis as shown in Appendix 10.

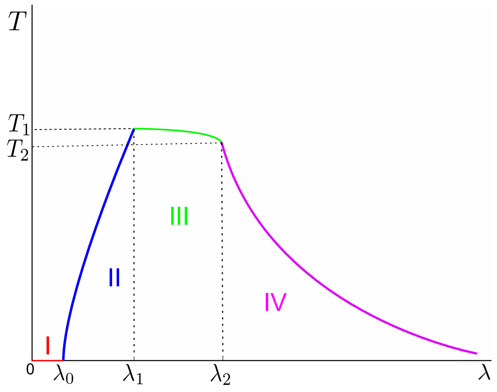


Fig. 3. Example of the optimal TTL value T for a given key at a given edge server vs. its arrival rate λ.

$\lambda^m_{k'} \geq \frac{\alpha d^m_{k'}}{(1-\alpha)c^m_{k'}}$ and $\lambda^m_{k'+1} < \frac{\alpha d^m_{k'+1}}{(1-\alpha)c^m_{k'+1}}$. From (1) and Theorem 3.1, we immediately have the following corollary.

Corollary 3.2. *Let B_m be the expected number of keys at edge server m. Then,*

$$B_m = \sum_{k=1}^{N_m}(1-m_k) = \sum_{k=1}^{k'}\left(1-\sqrt{\frac{\alpha d^m_k}{2(1-\alpha)c^m_k\lambda^m_k - \alpha d^m_k}}\right). \tag{6}$$

We next discuss how the TTL adapts to different factors: stream arrival rate, delay/traffic unit cost, and the weighting factor α.

3.1 TTL adaptation to arrival rate

Let λ be the arrival rate of a given key at a given edge, c be its unit traffic cost, d be its unit delay cost, and T be the optimal TTL derived for that key using Theorem 3.1. Figure 3 shows how optimizer *automatically* adjusts the TTL T of the key in accordance to its arrival rate λ. The optimized TTL aggregator exhibits different behaviors within each of four different ranges for the arrival rate λ as we describe below.

Region 1: $0 \leq \lambda \leq \lambda_0$. In this region, the arrival rate λ is too low for any aggregation to occur and the TTL T is set to zero. That is, the TTL aggregator “streams” the incoming records directly to the center.

Region 2: $\lambda_0 \leq \lambda \leq \lambda_1$. In this region, the arrival rate λ is low but large enough to allow aggregation. The TTL in the region is a concave increasing function of the arrival rate.

Region 3: $\lambda_1 \leq \lambda \leq \lambda_2$. In this region, the arrival rate is high enough that TTL can start to be lowered while still achieving the traffic benefits of aggregation. The TTL in this region is a concave decreasing function of the arrival rate.

Region 4: $\lambda \geq \lambda_2$. In this region, the arrival rates are very high and TTL can be lowered more significantly and still maintain the traffic benefits of aggregation. The TTL in this region is a convex decreasing function of the arrival rate.

The transitions points between the regions and the corresponding TTL values shown in Figure 3 can also be determined: $\lambda_0 \triangleq \frac{\alpha d}{(1-\alpha)c}$, $\lambda_1 \triangleq \frac{(2+\sqrt{2})\alpha d}{(1-\alpha)c}$, while λ_2 can be derived numerically. In practice, the arrival rate λ can be computed directly from the incoming data stream (see Section 4).

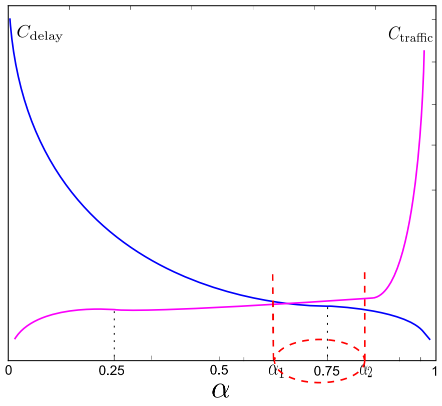


Fig. 4. Example of stable operating region. The values of α inside the dashed red circle are stable operating points, that correspond to relatively small delay and traffic that do not change too much with small changes in α.

3.2 TTL adaptation to unit cost

We next consider how TTL is affected by the relative delay and traffic unit costs (d/c). It is clear from (5) that the optimal timer T is decreasing in d/c, for a given key at a given edge with a given arrival rate λ, and α. This is intuitive, as the ratio d/c increases, the cost of delay has a larger impact than the cost of traffic, hence, to achieve the optimal tradeoff, it is intuitive to decrease the TTL T such that C_{delay} is decreased, and C_{traffic} is increased. The converse holds when d/c is decreased.

As we described in the real-world examples in Section 1, there are can be different sets of delay requirements for different customers of an analytics service. For instance, the keys that are aggregated for a real-time marketing campaign or for QoS monitoring are more delay sensitive than keys aggregated for a brand awareness campaign or for audience analytics. The unit delay cost quantifies the delay sensitiveness of the key and represents the financial cost of an additional unit of delay for the analytics customer. Thus, higher unit delay costs represent greater delay sensitivity and vice versa. On the other hand, the unit traffic cost could represent the bandwidth cost incurred by the analytics provider to transmit an additional data record.

3.3 TTL adaptation to α

The weighting factor α is an input parameter to the system, and the system operator can choose different values of α to balance the impact of delay and traffic on the total cost. It is clear from (5) that the optimal timer T is deceasing in α, for a given key at a given edge with a given arrival rate λ in regions II, III, and IV in Figure 3, and given d and c values. Furthermore, the threshold λ_0 is increasing in $\alpha \in (0, 1)$, with its value satisfying $(0, \infty)$. For the two extreme cases, when $\alpha \to 0$, $\lambda_0 \to 0$ and $T \to \infty$, this corresponds to batching; when $\alpha \to 1$, $\lambda_0 \to \infty$ and $T \to 0$, this corresponds to streaming.

One possibility is to automatically derive the α value(s) that achieve(s) the "best" delay-traffic tradeoff. To do so, we characterize the *stable operating region* across all the values of α. Intuitively, our goal is to find a "sweet spot" operating region where both delay and traffic are relatively small. Furthermore, for any point in this region, the value of delay and traffic do not change too much around this point, i.e., the derivatives of delay and traffic are bounded at this point. Figure 4 illustrates the stable operating region for a pair of delay-traffic cost curves against α.

Choosing α in practice. As explained above, α is a system-wide knob which the analytics provider can set to tradeoff delay and traffic costs across all consumers hosted on the analytics system. Using the real-life examples discussed in Section 1, an analytics system that hosts customers of the QoS monitoring service may choose a higher value of α, say $\alpha > 0.8$, to emphasize delay sensitivity of the service. While an analytics system hosting consumers of the Audience Analytics service may chose a lower value of α, say $\alpha < 0.2$, to emphasize traffic cost sensitivity. An analytics system that hosts a mix of delay and traffic cost sensitive services may choose a neutral value of α and capture the varying requirements through the per-key unit costs for delay and traffic.

The above examples are meant to help us understand that the choice of α is very much dependent on the type of services hosted by the analytics system. Having α as a user input adds to the flexibility of the overall optimization framework and ultimately, shows the wide applicability of the proposed optimization framework.

4 APACHE FLINK IMPLEMENTATION

In this section, we provide details of our implementation of the proposed TTL aggregation and its optimization framework in Apache Flink [9], a popular stream processing engine. Apache Flink follows the *dataflow* model [4] as its computation model. In the *dataflow* model, data streams enter continuously into the system from various data sources and are transformed by a set of stream operators. We introduce two new operators - *TTLAggregation* operator, which supports TTL aggregation and *TTLAggregationOptimized* operator which incorporates the proposed optimization framework. We first give details of these operators' API for any user to use them in their application. Then we give the details of the actual implementation of these operators in Apache Flink.

4.1 TTL Aggregation operators

***TTLAggregation* operator.** The *TTLAggregation* operator allows the user to perform the proposed TTL aggregation as explained in Section 2. The operator API is similar to that of the existing window operators like Tumbling windows and Sliding windows [35]. *TTLAggregation* operator takes as input the TTL value and the aggregation function to be performed on the data stream. For example, the user can use the *TTLAggregation* operator as follows:

```
input
   .keyBy(new KeySelectorFunction())
   .process(new TTLAggregation(TTL,
   new UserDefinedAggregateFunction()
   ));
```

Here, `TTL` is the TTL value to be used for TTL aggregation. It can be the same (a single floating point value) or different (stored as a map, `Map<Key, TTL>`) across keys. Additionally, users can also specify a default TTL as a second argument which will be used for any new keys in the data stream. In the scenarios where the unique number of keys are very high, the user may want to partition the keys into a smaller number of classes and instead provide the TTL for each class. This essentially means adding another level of keying in the data stream.

***TTLAggregationOptimized* operator.** The advantage of the proposed *TTLAggregation* operator over the existing window operators such as Tumbling window operator is that it can benefit from the optimization framework proposed in this paper. It can help the user achieve the *delay-traffic* tradeoffs achieved by the proposed optimization framework. To this end, we introduce another operator called as *TTLAggregationOptimized* which uses the optimization framework proposed in this paper to compute the TTL value for every key. The user can also choose between per-key optimization or global optimization in which the entire data stream is considered to be belonging to a single global key. For example, the user can use per-key optimization in the following way:

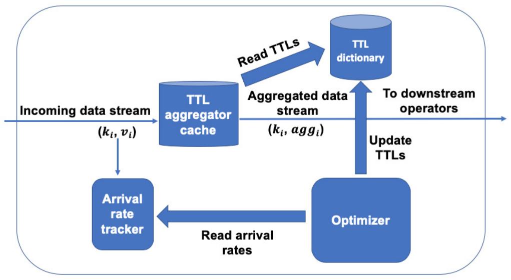


Fig. 5. Implementation design for *TTLAggregationOptimized* operator in Apache Flink.

```
input
   .keyBy(new KeySelectorFunction())
   .process(new TTLAggregationOptimized(PerKey=True, α,
      DelayCost, TrafficCost, InitialTTL, UpdateInterval,
      new UserDefinedAggregateFunction()
   ));
```

Here α, `DelayCost` and `TrafficCost` are user-defined tradeoff parameter, unit delay cost and unit traffic cost respectively, as discussed in Section 3. This operator allows the unit delay cost and unit traffic cost to be same (single floating point value) or different (`Map<Key, UnitCost>`) across all keys, allowing multi-class differentiation.

Adaptive TTL optimization. In a real system, the arrival rates of incoming data streams will vary over time. Therefore, we implement an adaptive TTL optimization where TTLs are re-computed periodically based on current key arrival rates. The `UpdateInterval` argument allows the user to recompute the TTLs after every `UpdateInterval` time units. `InitialTTL` is the TTL value assigned to each key until the system has sufficient data to measure its arrival rate.

Note that the proposed TTL operators can also be implemented in other stream processing engines such as Spark Streaming [36] and Apache Heron [24] which we leave as part of the future work. In Section 5, we also show how our TTL-based optimization framework can be used to optimize delay-traffic tradeoff for traditional windowed group aggregation.

4.2 Prototype

The design of *TTLAggregationOptimized* operator is shown in Figure 5. It has the following four components:

TTL dictionary. It stores the up-to-date TTL value for each key.

TTL aggregator cache. It functions exactly as explained in Section 2. It receives the incoming data stream, performs TTL aggregation, stores the most up-to-date aggregated value for every key and sends the aggregated records to downstream operators. If the incoming record's key does not exist in the TTL aggregator cache, the key is inserted into the TTL aggregator cache and the key's timer value is set to its current TTL value, fetched from the TTL dictionary. If the key already exists in the TTL aggregator cache, the aggregated value of the key is updated with the current record's value. Whenever the key's timer expires, the TTL aggregator cache evicts the key and sends the aggregated record to the downstream operators.

Arrival rate tracker. The Arrival rate tracker monitors the incoming data stream. It maintains the most up-to-date arrival rate of each key and updates the arrival rate of a key whenever it receives a new record for that key. The average arrival rate for each key is computed using exponential smoothing [8] (with *smoothing factor* = 0.9 in our implementation). Note that more sophisticated methods of arrival rate prediction can be used but the investigation of these methods is beyond the scope of this paper and are left as part of future work.

Optimizer. The optimizer is the main component implementing the entire optimization framework proposed in Section 3. The optimizer runs an adaptive algorithm that periodically retrieves the most up-to-date arrival rates for all the keys from the Arrival rate tracker and recomputes the TTL values for the keys for which the change in arrival rate is above a certain threshold. The TTL values are computed using Equation (5). The recomputed TTL values are saved into the TTL dictionary for subsequent accesses. Note that the TTL computation for each key involves a single mathematical formula whose computation overhead is negligible.

TTLAggregation operator has a similar design to the *TTLAggregationOptimized* operator. The only difference is the absence of the Arrival rate tracker component and the way in which TTL values are computed. The *TTLAggregation* operator directly takes the TTL values as an input from the user and does not modify them during the course of the query execution.

5 EMPIRICAL EVALUATION

5.1 Experimental setup

We consider the hub-and-spoke system topology where there is a central server connected to multiple edge servers. In this setup, a Flink cluster runs on each of the central and edge server sites. Each of the edge sites continuously ingests a separate incoming data stream, performs aggregation using the *TTLAggregationOptimized* operator (Section 4.1) and then forwards the aggregated data to the central server. The central server receives the data from all the edge servers, performs a final aggregation and then saves the final results into a database. Additionally, we also run a data streamer locally on each of the edge sites to generate raw input data streams which are sent to the respective edge clusters. We now describe each of the above mentioned components in detail.

Data streamer. It replays timestamped records from a trace file and can speed up or slow down record replay to explore different stream arrival rates. Each record has three components: *<arrival timestamp, key, value>*. The data streamer sends each record *<key, value>* to the Flink job running at the edge according to its timestamp.

Edge site. The Flink job running on the edge cluster receives data records from the data streamer and assigns an arrival timestamp to each record. Then the records are partitioned based on keys using the `keyBy` operator. The partitioned records are then forwarded to the `TTLAggregationOptimized` operator. This `TTLAggregationOptimized` operator performs the user specified aggregation function using the proposed TTL aggregation mechanism (Section 2) and then forwards the aggregated records to the output sink operator. The output sink operator assigns eviction timestamp to every record and sends the record to the TCP socket $Socket_{center,in}$ listening at the central server site.

Central site. The center continuously receives the incoming partially aggregated records from the edge, assigns them an arrival timestamp, performs the final aggregation and updates the database with the most recent aggregated values.

We demonstrate the effectiveness and practicality of our algorithms through a real deployment of the hub-and-spoke system topology on two testbeds:

AWS EC2 testbed. We use 6 AWS EC2 (t2.xlarge instance type) geographically distributed sites for our experiments. We run the central server in California region while the edge servers are deployed in Virginia, Oregon, London, Mumbai and Seoul. The data streamer runs locally on each of the edge server sites.

Emulated testbed. We also run our experiments on a local 12-node testbed which emulates the WAN bandwidth characteristics measured across the AWS EC2 sites. Each of the central server and edge servers run on a cluster of two machines each. Each machine is a 6-core Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz. All the machines are connected by a 1 Gbps ethernet. We emulate WAN characteristics using Linux TC utility.

Evaluation metrics. We use the following metrics:

• *Delay:* We measure the edge aggregation delay to be the time spent by a record at the edge. This is computed by taking the difference between the eviction timestamp and the arrival timestamp of the record at the edge. As discussed earlier, we mainly focus on the edge aggregation delay here. We also compute the edge-to-center network delay and show it in results as appropriate.

• *Traffic:* We define traffic to be the number of records transmitted per second over the WAN.

5.2 Datasets and queries

Datasets. We use two datasets to evaluate our proposed approach: a collection of anonymized beacon logs from Akamai's download analytics service, and the real tweet data from Twitter.

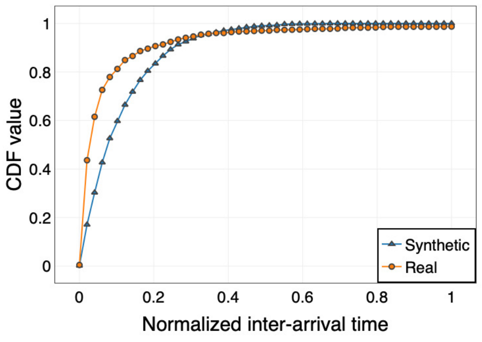
Akamai [25] operates a large geographically distributed content delivery network. It runs real-life analytics services for a variety of purposes. One such service is the download analytics service [1] which is consumed by the content providers for tracking important metrics such as the unique number of users downloading a particular content, the location from where an item is downloaded, the type of device used for downloading, the download performance experienced by the user, the number of successful complete downloads and so on. These metrics are collected from a software called as Download Manager [2] that is installed on millions of user devices such as mobiles, desktops, tablets, laptops, etc. This download manager is typically used for downloading softwares, software updates, music, videos, games, security updates, etc. The download manager running on the user device logs information about its downloads to the geographically distributed edge servers using beacons[6]. These beacons contain anonymized information about the download start time, url, content size, number of bytes downloaded, user's ip, user's network, user's geography, server's network and server's geography. The beacon logs were collected for an entire one month. *We normalize data sizes, traffic sizes, time durations etc, for the purpose of keeping confidentiality. We use the maximum value of these metrics that are plotted as the denominator for normalization, so that points have co-ordinates in the range* $[0, 1]$.

The real tweet data from Twitter was collected using publicly available Twitter Streaming APIs [31] in December 2015. It consists of approximately 4 million tweets per day. The Twitter Streaming APIs only give a sample of the actual Twitter workload. Hence, we scaled the playback rate to around 6000 tweets per second to emulate the actual tweet rate [32]. Each tweet contains information such as the username, user location, language of the tweet, actual tweet contents, user gender, age, country, etc. Each tweet record is on an average 450 bytes which leads to a tweet stream rate of around 20 Mbps.

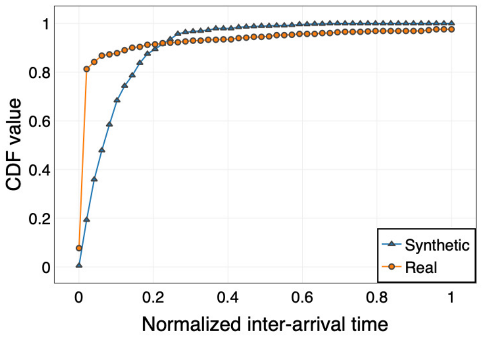
For both the datasets, we distribute the data records across the edge server sites based on their geographic information.

Queries. For the Akamai workload, we compute the *Sum*, *Max* and *HyperLogLog* [13] aggregation for a very common query which groups by content provider id, the user's country code, and the url accessed. The number of unique keys in this query are of the order of 10^6. The Sum aggregation computes the total number of bytes successfully downloaded for each key, the Max aggregation computes the largest successful download size for each key, and the HyperLogLog aggregation is used to approximate the number of unique client IP addresses for each key. Due to space constraints, we only show the results corresponding to the Sum aggregation queries, but similar results hold for

[6]A beacon is a HTTP GET request by the Download Manager for a small GIF containing the reported values in its url query string.



(a) Representative key 1.



(b) Representative key 2.

Fig. 6. *CDF of inter-arrival times for real Akamai trace and synthetic trace for two representative keys.*

other queries and aggregation operators. For Twitter data, we show results for the popular word count query which computes the frequency of words appearing in tweets grouped by location. Similar results hold for other Twitter queries such as the trending topics query which computes the frequency of tweets grouped by location, language and topic but they are not shown due to space constraints.

5.3 Empirical results

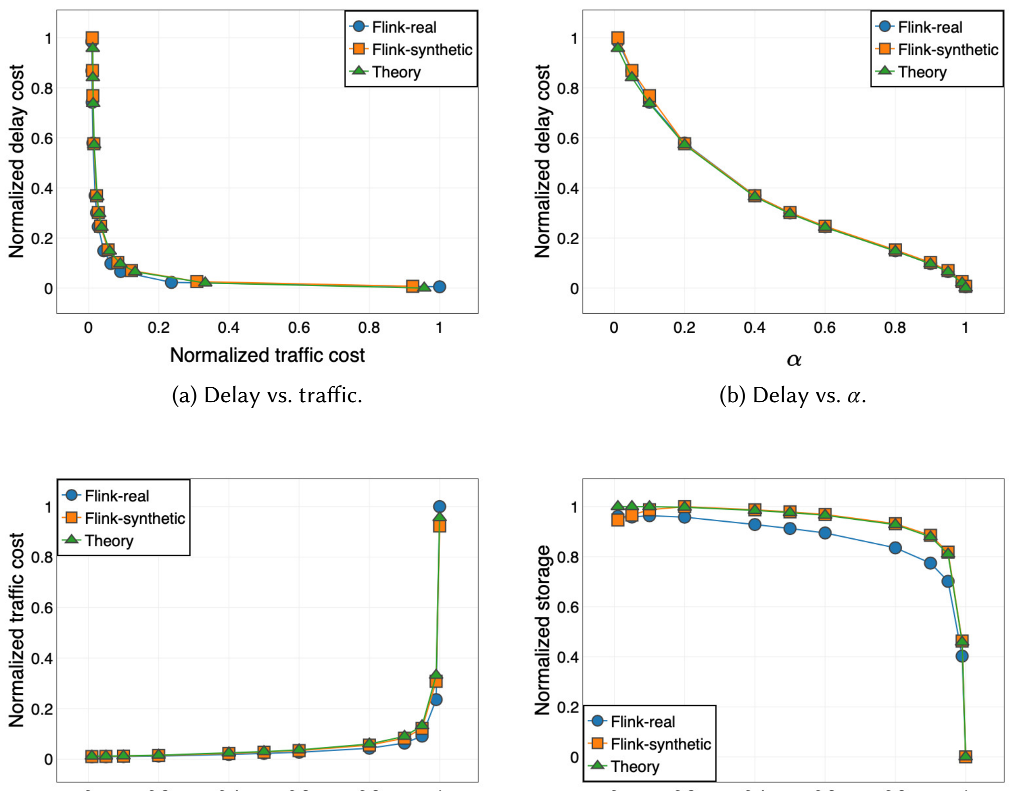
We first show the key results of our evaluation by running experiments using the Akamai trace on the AWS EC2 testbed.

Characteristics of the real-world Akamai trace. We first show that the real-world Akamai trace does not have strict Poisson arrivals for any key. To that end, we generate a synthetic trace based on the real Akamai trace, where each key follows Poisson arrivals with the same average arrival rate as in the Akamai trace. We analyzed the distribution of inter-arrival times for the corresponding keys from the real and synthetic trace and found them to be visibly different as depicted in Figure 6. Additionally, we performed the Two Sample Kolmogorov-Smirnov test (KS test) for a quantitative comparison of the distribution of inter-arrival times for the corresponding keys in real and synthetic traces for Akamai. We used a P-value of 0.05 for rejecting the null hypothesis that the two samples were drawn from the same distribution. None of the keys passed the KS test, showing that the real Akamai trace does not have strict Poisson arrivals for any key.

Comparison of theoretical and empirical results. We now show that despite the fact that the real-world traces, such as the Akamai trace, may not be strictly Poisson, our proposed theoretical optimization model with Poisson assumptions works well in practice. To that end, we compare the theoretically computed average delay, traffic, and storage costs (calculated using Equations (2), (3), and (1)) to the empirically computed delay, traffic, and storage costs. In particular, we compare three results: 1) empirical costs for the real Akamai trace, 2) empirical costs for the synthetic Poisson trace with same key arrival rates as the Akamai trace as described earlier, and 3) theoretical costs for the same synthetic trace. We vary the tradeoff parameter α from 0 to 1 and for each α, we compute all three results. We use $d = 0.01, c = 1$ for all the keys at all the edges.

Figure 7 compares the curves for all three cases. We can see that the theoretical costs match very well with the empirical costs for the synthetic trace while the empirical costs for the real Akamai trace differ slightly more with the other two.

Another important takeaway from Figure 7 is the variation of delay and traffic costs as α goes from 0 to 1. Our proposed approach can cover the entire delay-traffic tradeoff curve with batching



(a) Delay vs. traffic.

(b) Delay vs. α.

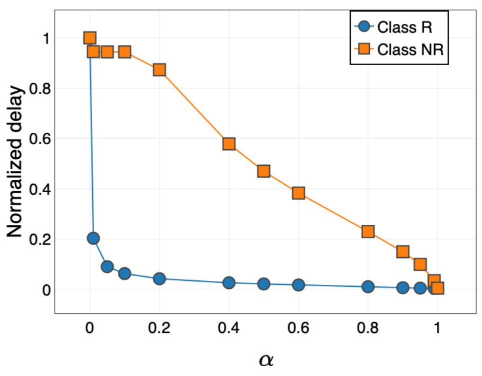
(c) Traffic vs. α.

(d) Storage vs α.

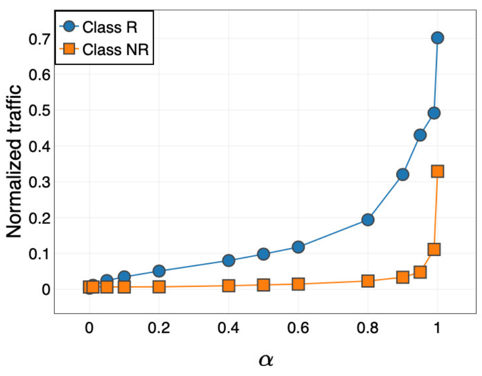
Fig. 7. *Comparison between theory and experimental results of delay-traffic tradeoff for Akamai trace.*

on the left end and streaming on the right end. Depending on the delay-traffic tradeoff requirement of the application, the user can choose an appropriate α. For instance, if the user is looking for the "best" delay-traffic tradeoff, an α value in the range 0.8-0.9 would work well in this experiment. Here, the "best" delay-traffic tradeoff is able to *simultaneously achieve 90% reduction in delay as compared to batching and 94% reduction in traffic as compared to streaming.* Given the close match between theoretical and empirical results, a key point here is that this "best" (or stable) operating region *can be derived theoretically* (as discussed in Section 3.3), rather than through trial-and-error empirically, as is the common practice today.

Multi-class traffic differentiation. Next, we show the effect of the variation in d_k^m/c_k^m ratio on T_k^m and consequently on the delay-traffic tradeoff. We consider two classes of keys: high priority keys (called as class R) having $d_k^m = 1, c_k^m = 1$ and low priority keys (called as class NR) having $d_k^m = 0.01, c_k^m = 1$. Here, class R has a relatively higher delay-traffic unit cost ratio (d_k^m/c_k^m) compared to class NR, and thus, a lower tolerance for delay, even at the expense of higher traffic. We can see from Figure 8 that the average delay is lower for high priority keys while the average traffic is higher for high priority keys. This result is in accordance with the theory in Section 3.2. This differentiation is useful in the scenarios where some keys are less delay tolerant than the



(a) Delay vs. α.



(b) Traffic vs. α.

Fig. 8. *Delay-traffic tradeoffs for multi-class traffic. The delay (resp. traffic) for class R (*$d_k^m/c_k^m = 1$*) is always lower (resp. higher) compared to that for class NR (*$d_k^m/c_k^m = 0.01$*).*

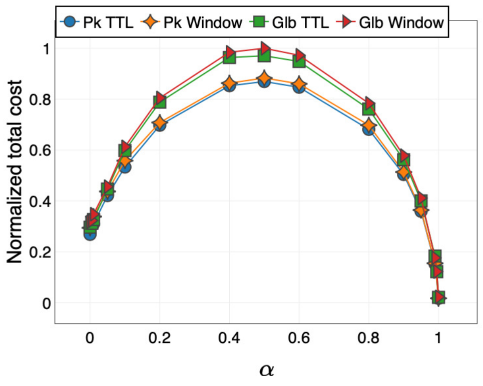


Fig. 9. *Comparison of total cost for per-key (Pk) and global (Glb) approaches for both TTL-based aggregation (TTL) and windowed grouped aggregation (Window).*

others. A typical example of this would be a QoS monitoring data stream which sends the video delivery statistics to the center for all the users. If there are both paid users and unpaid users in the incoming data stream, then the paid users' statistics are more critical and must reach the center sooner than that of the unpaid users. In this case, the paid users' keys would be assigned lower delays compared to the unpaid users' keys.

We now show other results of our evaluation by running experiments on the emulated testbed.

Application of TTL-based optimization to windowed grouped aggregation. Here, we show how our proposed TTL-based optimization framework can be used to optimize windowed group aggregation, where the optimal window size is derived by our optimization framework.

We compare the delay-traffic tradeoff for our TTL-based aggregation operator to a windowed grouped aggregation operator, *that uses the TTL values derived from our optimization framework* as its window size. We also consider two types of windowing (and TTL-based) approaches. One is *per-key* windowing where each key has its own window size, i.e., the windows are *unaligned*

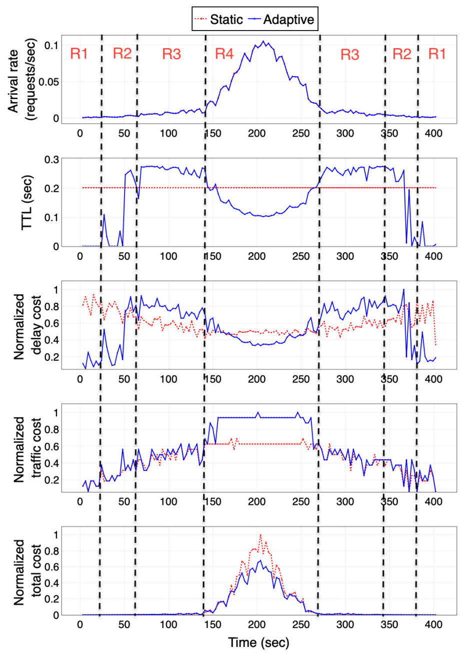


Fig. 10. *Comparison of static TTL and adaptive TTL.*

in stream processing terminology. The second is *global* windowing where all keys have *aligned* windows of same size. For computing the TTL for the global windowing case, instead of considering the individual arrival rates of every key, we compute the average of the arrival rates of all the keys and use this average as the common arrival rate for all the keys. We vary the tradeoff parameter α from 0 to 1 and we use $d = 0.01, c = 1$ for all the keys.

Figures 9 shows the total cost comparison for windowed grouped aggregation and TTL-based aggregation approach. We see that the results match closely for the per-key windowing as well as for the global windowing. At the same time, the total cost for per-key TTL-based aggregation approach can be up to 10% less than that of the global TTL-based aggregation approach. Similar trend is seen for per-key vs. global windowed grouped aggregation as well. From these results, we conclude that the optimization model proposed in this work can be used for achieving a good delay-traffic tradeoff in windowed grouped aggregation (*if* the window sizes are tunable and can be used with per-key windowing).

Adaptive TTL optimization. In practice, the arrival rates of the keys in the incoming data stream may not be known a priori, and may also change over time. Thus, it is important to recompute the TTLs periodically so that the TTLs reflect the current arrival rates. To show the performance of our proposed approach in such scenarios, we consider a synthetic trace in which the arrival rates keep on changing continuously (first increase for a certain time and then decrease). The record arrivals follow Poisson distribution. We consider two approaches:

- *Static TTL approach.* In this approach, the average arrival rates of the keys are computed using the entire trace at the beginning of the experiment (hence, this is the best static approach). We use this pre-computed arrival rate to compute the TTL value for each key and use this TTL value for the entire processing of the synthetic trace.
- *Adaptive TTL approach.* In this approach, we start from an initial TTL and recompute the TTL periodically online using the most up-to-date arrival rate of the keys, as discussed in Section 4.2.

Figure 10 compares the static TTL approach with the adaptive TTL approach. Due to space constraints, we only show the results of one key. Results for other keys also follow the same trend. Here, we fix $\alpha = 0.6, c = 1, d = 1$. We can see that as the arrival rate varies, the TTL value also keeps on changing taking into account the most up-to-date arrival rate. An interesting point to note here is that the TTL varies in accordance with the theory proposed and explained in Section 3.1. Section 3.1 explains that as the arrival rate increases, the TTL goes through four regions: Region 1, where the TTL is zero because of very low arrival rate, Region 2, where the TTL is a concave increasing function of the arrival rate, Region 3, where the TTL starts decreasing and Region 4, where the TTL decreases significantly. These four regions are shown as R1, R2, R3 and R4 in Figure 10. Note that there is some noise in the initial portion and the last portion of the trace due to the arrival rate being too low, which sometimes results in bursty arrivals in the adaptation intervals. The more interesting portion is the middle portion where the arrival rate increases significantly. Here, we see that initially TTL also increases with the arrival rate (Region 2), but beyond a point, it switches to decreasing with increasing rate (Region 3), and continues to decrease as the arrival rate increases (Region 4). Once the arrival rate starts decreasing, we see a reverse transition through the different regions.

Based on the variation in TTL, the instantaneous delay cost, traffic cost and total cost also vary accordingly. We see that at low arrival rates, the total cost for both the static and adaptive approaches is about the same (close to 0), but as the rate increases, Adaptive achieves lower total cost (up to 31% near the peak arrival rate). Looking at the delay and traffic curves, we find that the TTL is tuned to lower delay aggressively at the expense of higher traffic during high rate periods. This is primarily because of the parameter values of α, c and d, which prefer lower delay compared to traffic. For other values of these parameters, the optimization framework will still try to achieve low total cost, though the desired delay-traffic tradeoff point may be different.

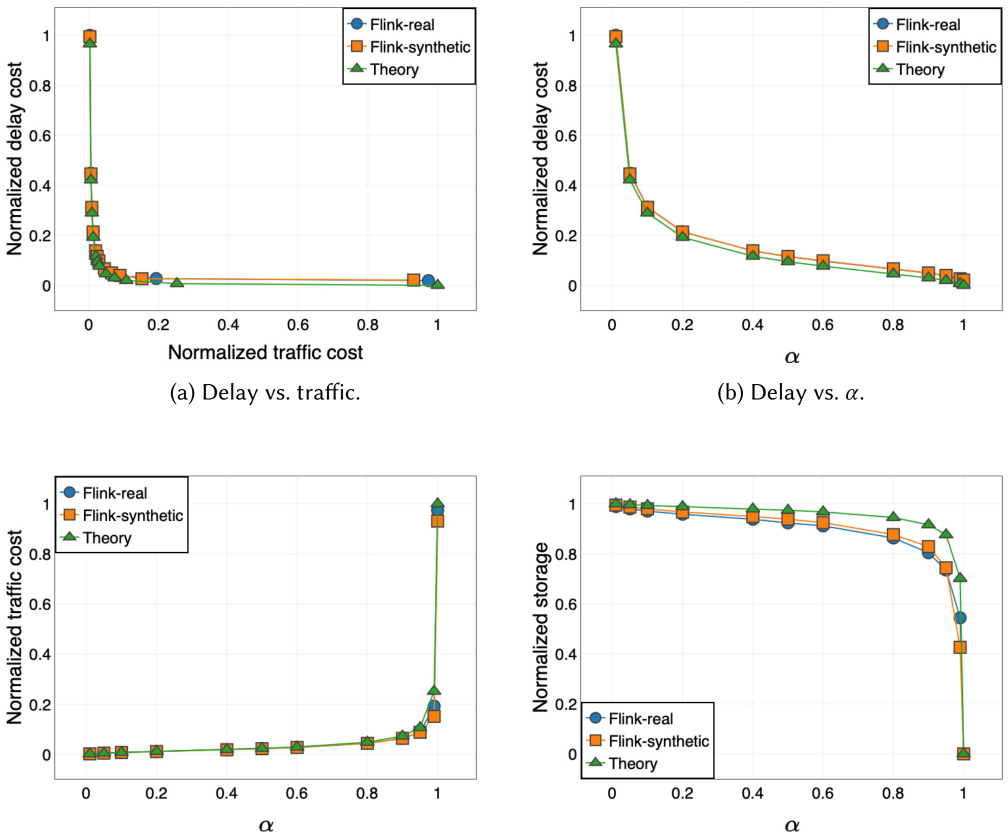
Twitter trace results. We next show one key result of our evaluation by running experiments using the Twitter trace on the emulated testbed. Due to space constraints, we omit other results but the conclusions remain the same. As in the case of Akamai trace, we again confirm that the real-world Twitter trace does not have strict Poisson arrivals for any key. For this, we generate another synthetic trace based on the real Twitter trace where each key follows Poisson arrivals with the same average arrival rate as in the Twitter trace. The KS-test (with P-value of 0.05) comparing the distribution of the inter-arrival times for the corresponding keys in the real and synthetic traces fails for all the keys, indicating that the real Twitter trace does not have Poisson arrivals for any key.

Here, we again show that the proposed optimization model works well in practice by comparing our empirical results to the theoretical results for the Twitter trace. Similar to Figure 7, we consider three results: 1) empirical costs for the real Twitter trace, 2) empirical costs for the synthetic trace and 3) theoretical costs for the synthetic trace. We vary the tradeoff parameter α from 0 to 1 and for each α, we compute all three results.

The results are shown in Figure 11. We can see that in this case also, the theoretical costs match very well with the empirical costs. The "best" delay-traffic tradeoff in this case can be achieved by choosing α in the range 0.4-0.5 where both delay and traffic are small in comparison to the batching delay and streaming traffic.

6 EXTENSIONS

Here, we show how our framework can be extended to explore TTL aggregators in other types of problems. We consider two particular problems. One is to minimize the delay cost with a traffic constraint, the other is to minimize the traffic cost with a delay constraint.



(a) Delay vs. traffic. (b) Delay vs. α.

(c) Traffic vs. α. (d) Storage vs. α.

Fig. 11. *Comparison between theory and experimental results of delay-traffic tradeoff for Twitter trace.*

6.1 Minimizing delay (traffic bound)

Since the bandwidth for the edge-to-center WAN is usually limited, we consider the problem to minimize the delay subject to a traffic constraint. Our goal is to find the optimal vector of timers $\boldsymbol{T} = \{T_k^m\}_{k\in\mathcal{N}_m, m\in\mathcal{M}}$ to minimize the total delay cost C_{delay} incurred across all keys and edges, such that the traffic over all keys on each edge is bounded, i.e.,

$$\begin{aligned}\min \quad & C_{\text{delay}} \\ \text{s.t.} \quad & C_{\text{traffic}}^m \le W_m, \quad m = 1, \cdots, M, \\ & T_k^m \le T_{k,\max}^m, \quad k = 1, \cdots, N_m, \quad m = 1, \cdots, M,\end{aligned} \tag{7}$$

where C_{traffic}^m is the traffic caused by all keys on edge m. Using Theorem 2.1, we can write C_{traffic}^m as the sum of the delay cost of all keys over edge m, i.e., $C_{\text{traffic}}^m = \sum_{k=1}^{N_m} \frac{c_k^m \lambda_k^m}{1+\lambda_k^m T_k^m}$. W_m is a constant, e.g., average bandwidth limit on traffic for edge m. $T_{k,\max}^m$ is the maximal delay that key k can tolerate, which is a constant based on the application.

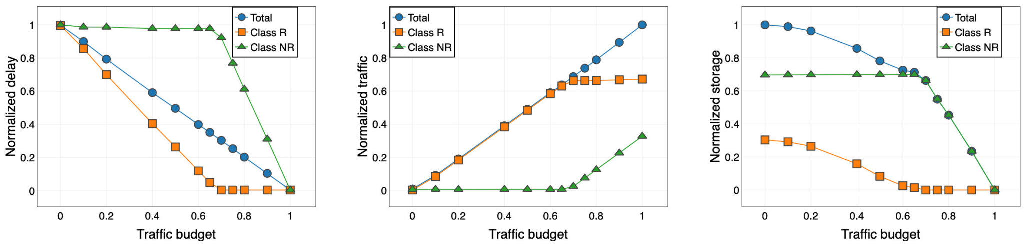
THEOREM 6.1. *The optimal timers (TTLs) that minimize the expression for C_{delay} in Equation (7) satisfy*

$$T_i^m = 0, \quad i = 1, \cdots, j,$$
$$T_i^m = T_{i,max}^m, \quad i = j+1, \cdots, N_m, \tag{8}$$

where we assume that the delay costs per content are sorted as $d_1^m \geq d_2^m \geq \cdots \geq d_{N_m}^m$, and j satisfies

$$\left\lceil \sum_{k=1}^{j} c_k^m \lambda_k^m + \sum_{k=j+1}^{N_m} \frac{c_k^m \lambda_k^m}{1 + \lambda_k^m T_{k,max}^m} \right\rceil = W_m, \tag{9}$$

where $\lceil \cdot \rceil$ is the ceiling function.



(a) Delay vs. traffic budget. (b) Traffic vs. traffic budget. (c) Storage vs. traffic budget.

Fig. 12. *Minimize delay with a constraint on traffic.* Keys with higher unit delay cost (d_k^m) are always given preference over the keys with lower unit delay cost in the assignment of available traffic budget.

Empirical results. Intuitively, Theorem 6.1 tries to assign the available traffic budget (called constraint in the theorem) to the keys having higher unit delay costs as compared to others. To showcase how this theorem works in practice, we run a simple empirical analysis on the emulated testbed using the Akamai trace. In this experiment, we consider two classes of keys: high priority keys (class R) having higher $d_k^m (= 1)$ and low priority keys (class NR) having lower $d_k^m (= 0.01)$. We use $c_k^m = 1$ for all the keys. We vary the available traffic budget from zero to the minimum traffic budget required for streaming all the keys without any delay.

From Figure 12, we can see that initially, when the available traffic budget is very small, the model assigns the complete budget to high priority keys. As the available traffic budget increases, the model first assigns the available budget to the high priority keys and then assigns the remaining portion of the budget to the low priority keys. Therefore, we see that first the average delay goes on decreasing for the high priority keys until it becomes close to zero. Till this point, the average delay for the low priority keys remains high since they have not been assigned any traffic budget. After this point, the average delay for the low priority keys starts decreasing as they also start getting assigned some portion of the traffic budget. Correspondingly, the traffic for high priority keys keeps increasing with the increase in the available traffic budget until almost all the high priority keys are sent without any delay. Till this point, the traffic for the low priority keys is negligible since they have not been assigned any traffic budget. After this point, the traffic for the low priority keys starts increasing as they also start getting assigned some portion of the budget.

6.2 Minimizing traffic (delay bound)

Some applications have a stringent constraint on the delay between edge servers and the center, hence we consider the problem to minimize the traffic subject to a delay constraint. Our goal is to find the optimal vector of timers $\boldsymbol{T} = \{T_k^m\}_{k\in\mathcal{N}_m, m\in\mathcal{M}}$ to minimize the total traffic cost C_{traffic} incurred across all keys and edges, such that the delay over all keys on each edge is bounded, i.e.,

$$\begin{aligned} \min \quad & C_{\text{traffic}} \\ \text{s.t.} \quad & C_{\text{delay}}^m \le \Gamma_m, \quad m = 1, \cdots, M, \\ & T_k^m \le T_{k,\max}^m, \quad k = 1, \cdots, N_m, \quad m = 1, \cdots, M, \end{aligned} \tag{10}$$

where C_{delay}^m is the delay experienced by all keys on edge m. Using Theorem 2.1, we can write C_{delay}^m as the sum of the delay cost of all keys over edge m, i.e.,

$$C_{\text{delay}} = \sum_{m=1}^{M} \sum_{k=1}^{N_m} \frac{d_k^m \lambda_k^m T_k^m}{2} \left(1 + \frac{1}{1 + \lambda_k^m T_k^m}\right) \tag{11}$$

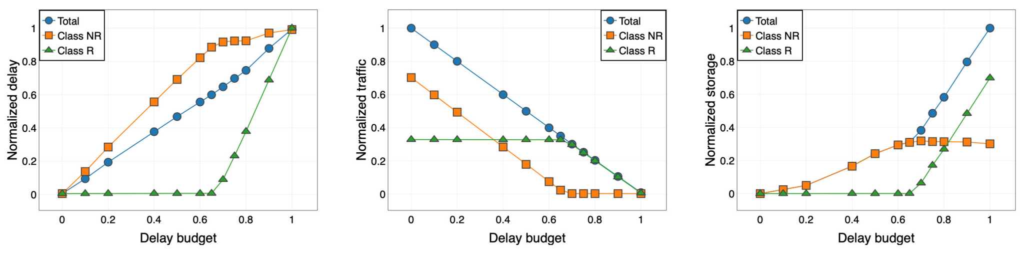
Γ_m is a constant, e.g., average delay limit for edge m.

Theorem 6.2. *The optimal timers (TTLs) that minimize the expression for $C_{traffic}$ in Equation (10) satisfy*

$$\begin{aligned} T_i^m &= T_{i,max}^m, \quad i = 1, \cdots, l, \\ T_i^m &= 0, \quad i = l+1, \cdots, N_m, \end{aligned} \tag{12}$$

where we assume that the traffic costs per content are sorted as $c_1^m \ge c_2^m \ge \cdots \ge c_{N^m}$, and l satisfies

$$\left\lceil \sum_{k=1}^{l} \left(\frac{d_k^m \lambda_k^m T_{k,max}^m}{2} \left(1 + \frac{1}{1 + \lambda_k^m T_{k,max}^m}\right) + d_k^m \lambda_k^m B_k^m \right) \right\rceil = \Gamma_m. \tag{13}$$



(a) Delay vs. delay budget. (b) Traffic vs. delay budget. (c) Storage vs. delay budget.

Fig. 13. *Minimize traffic with a constraint on delay.* The delay for class R ($d_k^m / c_k^m = 1$) is always lower in comparison to the delay for class NR ($d_k^m / c_k^m = 0.01$) while the traffic for class R is always higher in comparison to the delay for class NR.

Empirical results. We conduct a similar experiment to the traffic bound optimization case here using the Akamai trace on the emulated testbed. In this experiment, we again consider two classes of keys: high priority keys (class NR) having higher $c_k^m (= 1)$ and low priority keys (class R) having lower $c_k^m (= 0.01)$. We use $d_k^m = 0.01$ for all the keys. We vary the available delay budget from zero to the minimum delay budget required for batching all the keys. Figure 13 shows the results. We see a similar trend to that seen with the traffic bound case here, where the high priority class gets allocated its portion of the delay budget first, and the low priority class gets the remaining portion (if available). We note that our current formulation provides a strict prioritization of traffic or delay across classes based on their delay/traffic unit cost preferences. This formulation can be extended to achieve some fairness across different classes, to avoid starvation of lower priority classes. Such an extension is part of our future work.

7 RELATED WORK

Stream processing systems. A number of stream processing systems have been proposed in recent times [9, 23, 24, 36], which aim at supporting streaming data applications requiring low latency and high throughput. All of these systems work well in a single data center environment, where the available bandwidth is much more abundant compared to a wide-area setting. Our work focuses on achieving the delay-traffic tradeoff in a geo-distributed environment and can benefit from these systems for efficient stream processing at every individual edge or center. We have showcased the benefits of our approach by implementing a prototype in Apache Flink running at every edge and center.

Geo-distributed analytics. Much of the work in the area of geo-distributed analytics has focused on batch analytics [19, 27, 33], which optimizes query and placement of data and tasks to balance bandwidth usage and latency. On the contrary, our work focuses on streaming analytics, where latency is a critical metric.

JetStream [28] and AWStream [37] are wide-area streaming analytics systems. AWStream is an improvement over JetStream, but both systems focus on trading off accuracy with bandwidth consumption. Moreover, AWStream relies on offline empirical evaluation to choose the best tradeoff. Sana [21] is yet another wide-area streaming analytics system focusing on multi-query optimization in a wide-area environment. Our proposed work, on the other hand, provides a theoretical framework for trading off delay with traffic. Further, our online algorithm automatically tries to identify the optimal delay-traffic tradeoff. Some of the techniques proposed by these works, including quality degradation and multi-query optimization, are complementary to our work.

TTL caches. TTL caches have been employed in the Domain Name System (DNS) since the early days of Internet [22]. More recently, it has gained attention due to fact that a simple and tractable analysis can be modeled to mimic the behaviors of caching algorithms. [10, 11] first introduced the notion of characteristic time for LRU under IRM to show that TTL caches can be used to provide accurate estimates of the performance of large caches. The accuracy of TTL cache is theoretically justified under IRM [6] and stationary processes [20], and numerically verified under renewal processes [16]. Its performance in cache networks has been studied [6, 15, 26]. All of the prior work focus on using TTL caches for storing popular objects of various types. While our use of TTL for aggregation is novel, some of the prior work on the mathematical properties of TTL caches are relevant in our new context, allowing us to use the expression for miss rates derived in this literature.

Aggregation. Aggregation is an important operator in analytics and has been studied in the past in various contexts. Heintz et al. focused on delay-traffic tradeoff [18] as well as delay-accuracy tradeoff [17] in the context of windowed grouped aggregation in geo-distributed streaming analytics. Our work is different from both these works from two perspectives. First, we focus on continuous aggregation instead of windowed grouped aggregation. Second, we propose theoretically sound online algorithms for achieving the desired delay-traffic tradeoff as compared to the heuristic-based online algorithms proposed in both of these prior works. Amur et al. [5] studied grouped aggregation focusing on the design and implementation of efficient data structures for batch and streaming computation but did not consider delay as a performance metric which is critical in the geo-distributed setting. Aggregation has also been studied in sensor networks [29], where the goal is to gather and aggregate data in an energy efficient manner to enhance network lifetime. The goal of our work is different and is to achieve the desired delay-traffic tradeoff in a geo-distributed environment.

8 CONCLUSION

In this paper, we proposed a new TTL-based mechanism for aggregation that allows us to model and optimize the all-important delay-traffic tradeoff in wide-area streaming analytics. TTLs have been widely used in the context of caching frequently used objects, such objects range from DNS entries to web pages. However, its use for aggregating wide-area distributed data streams is novel. As we show with our Apache Flink implementation, TTL aggregators are easy to implement in the current stream processing frameworks. We also show that the TTL mechanism provides a set of knobs that can be used by the network operator to balance delay and traffic costs of a large geo-distributed analytics system in a manner that is predictable and theoretically well-founded. As future work, we plan to consider applications of the TTL approach to other types of wide-area stream processing.

9 ACKNOWLEDGMENTS

The authors thank the anonymous reviewers and our shepherd Ganesh Ananthanarayanan, for many constructive comments and suggestions that greatly improved the quality of this paper. The authors also thank Kshitij Tayal for his helpful comments on the formulations. This work was sponsored in part by NSF under Grants CNS-1717834 and CNS-1717179, as well as by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-16-3-0001.

REFERENCES

[1] Akamai Download Analytics solution. Accessed: 2018-10-29. https://www.akamai.com/us/en/multimedia/documents/product-brief/download-analytics-product-brief.pdf.

[2] Akamai Download Manager. Accessed: 2018-10-29. https://www.akamai.com/us/en/products/media-delivery/download-manager-overview.jsp.

[3] Akamai Media Analytics. Accessed: 2018-10-29. https://www.akamai.com/us/en/products/media-delivery/media-analytics.jsp.

[4] Tyler Akidau, Eric Schmidt, Sam Whittle, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. FernÃąndez-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, and Frances Perry. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. (2015).

[5] Hrishikesh Amur, Wolfgang Richter, David G. Andersen, Michael Kaminsky, Karsten Schwan, Athula Balachandran, and Erik Zawadzki. 2013. Memory-efficient Groupby-aggregate Using Compressed Buffer Trees. In *Proc. of ACM SOCC*.

[6] D. Berger, P. Gland, S. Singla, and F. Ciucu. 2014. Exact Analysis of TTL Cache Networks. *Performance Evaluation* 79 (2014), 2–23.

[7] Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin. 2014. Summingbird: A framework for integrating batch and online mapreduce computations. *VLDB* 7, 13 (2014), 1441–1451.

[8] Robert Goodell Brown. 1963. *Smoothing, forecasting and prediction of discrete time series.* Prentice-Hall Englewood Cliffs, N.J. 468 p. pages.

[9] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. (2015).

[10] H. Che, Y. Tung, and Z. Wang. 2002. Hierarchical Web Caching Systems: Modeling, Design and Experimental Results. *IEEE Journal on Selected Areas in Communications* 20, 7 (2002), 1305–1314.

[11] Ronald Fagin. 1977. Asymptotic Miss Ratios over Independent References. *J. Comput. System Sci.* 14, 2 (1977), 222–250.

[12] A. Ferragut, I. Rodríguez, and F. Paganini. 2016. Optimizing TTL Caches under Heavy-tailed Demands. In *Proc. of ACM SIGMETRICS*.

[13] Philippe Flajolet, ÃĽric Fusy, Olivier Gandouet, and et al. 2007. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *Proc. of IN AOFA*.

[14] N. C. Fofack, M. Dehghan, D. Towsley, M. Badov, and D. L. Goeckel. 2014. On the Performance of General Cache Networks. In *VALUETOOLS*.

[15] N. C. Fofack, P. Nain, G. Neglia, and D. Towsley. 2012. Analysis of TTL-based Cache Networks. In *VALUETOOLS*.

[16] M. Garetto, E. Leonardi, and v. Martina. 2016. A Unified Approach to the Performance Analysis of Caching Systems. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* 1, 3 (2016), 12.

[17] Benjamin Heintz, Abhishek Chandra, and Ramesh K. Sitaraman. 2016. Trading Timeliness and Accuracy in Geo-Distributed Streaming Analytics. In *Proc. of ACM SoCC*.

[18] Benjamin Heintz, Abhishek Chandra, and Ramesh K. Sitaraman. 2017. Optimizing Timeliness and Cost in Geo-Distributed Streaming Analytics. (2017).

[19] Chien-Chun Hung, Ganesh Ananthanarayanan, Leana Golubchik, Minlan Yu, and Mingyang Zhang. 2018. Wide-area analytics with multiple resources. In *Proc. of EuroSys*.

[20] B. Jiang, P. Nain, and D. Towsley. 2016. On the Convergence of the TTL Approximation for an LRU Cache under Independent Stationary Rrequest Processes. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* 3, 4 (2016), 20.

[21] Albert Jonathan, Abhishek Chandra, and Jon Weissman. 2018. Multi-Query Optimization in Wide-Area Streaming Analytics. In *Proc. of ACM SoCC*.

[22] J. Jung, A. Berger, and H. Balakrishnan. 2003. Analysis of TTL-based Cache Networks. In *IEEE INFOCOM*.

[23] KSQL: Streaming SQL for Kafka. Accessed: 2018-10-29. https://www.confluent.io/product/ksql/.

[24] Sanjeev Kulkarni, Nikunj Bhagat, Masong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Proc. Of ACM SIGMOD*.

[25] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. 2010. The Akamai Network: A Platform for High-performance Internet Applications. *SIGOPS Oper. Syst. Rev.* 44, 3 (2010), 2–19.

[26] N. K. Panigrahy, J. Li, F. Zafari, D. Towsley, and P. Yu. 2018. Optimizing Timer-based Policies for General Cache Networks. *Arxiv preprint arXiv:1711.03941* (2018).

[27] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. 2015. Low Latency Geo-distributed Data Analytics. In *ACM SIGCOMM*.

[28] Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek S. Pai, and Michael J. Freedman. 2014. Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area. In *Proc. of USENIX NSDI*.

[29] Ramesh Rajagopalan and Pramod Varshney. 2006. Data-aggregation techniques in sensor networks: a survey. 4 (2006), 48–63.

[30] Twitter Analytics. Accessed: 2018-10-29. https://business.twitter.com/en/analytics.html.

[31] Twitter Developer APIs. Accessed: 2018-10-29. https://developer.twitter.com/en/docs.

[32] Twitter usage statistics. Accessed: 2018-10-29. http://www.internetlivestats.com/twitter-statistics/.

[33] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. 2016. CLARINET: WAN-Aware Optimization for Analytics Queries. In *Proc. of USENIX OSDI*.

[34] Ashish Vulimiri, Carlo Curino, Philip Brighten Godfrey, Thomas Jungblut, Konstantinos Karanasos, Jitendra Padhye, and George Varghese. 2015. Wanalytics: Geo-distributed analytics for a data intensive world. In *ACM SIGMOD*. 1087–1092.

[35] Windows API in Apache Flink. Accessed: 2018-10-29. https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators/windows.html.

[36] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *ACM SOSP*. 423–438.

[37] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A. Lee. 2018. AWStream: adaptive wide-area streaming analytics. In *Proc. of ACM SIGCOMM*.

10 APPENDIX

Expected end-to-end delay: We characterize the average end-to-end delay that each key experiences as comprising the *network delay* due to key transmission from edge server to the central controller, as well as the *aggregation delay* caused by key aggregation at the edge server. W.l.o.g., we consider a key k from edge server m, $\forall m \in \mathcal{M}$. Using Theorem 2.1, the aggregation delay of key k at edge m, denoted by $D^m_{\text{a-delay-k}}$, can be derived as follows.

$$D^m_{\text{a-delay-k}} = \left(\frac{1}{1 + \lambda^m_k T^m_k} + 1 \right) \frac{T^m_k}{2},$$

where λ_k^m is the arrival rate of key k at edge m. Then the *total expected aggregation delay* for all keys from edge server m is

$$D_{\text{a-delay}}^m = \sum_{k=1}^{N_m} \lambda_k^m D_{\text{a-delay-k}}^m = \sum_{k=1}^{N_m} \left(\frac{1}{1 + \lambda_k^m T_k^m} + 1 \right) \frac{\lambda_k^m T_k^m}{2}, \tag{14}$$

and the *total expected aggregation delay cost* for all keys from edge server m is

$$C_{\text{a-delay}}^m = \sum_{k=1}^{N_m} d_k^m \lambda_k^m D_{\text{a-delay-k}}^m = \sum_{k=1}^{N_m} \left(\frac{1}{1 + \lambda_k^m T_k^m} + 1 \right) \frac{d_k^m \lambda_k^m T_k^m}{2}, \tag{15}$$

where d_k^m is a constant characterizing the average delay cost for key k.

The network delay is measured as the time between the key being flushed from the edge server to the time that the central hub receives it, which is a constant. Denote it as δ_k^m for key k from sever m. The *total expected network delay* experienced by all keys from edge server m is

$$D_{\text{n-delay}}^m = \sum_{k=1}^{N_m} \lambda_k^m \delta_k^m, \tag{16}$$

then *total expected network delay cost* experienced by all keys from edge server m is

$$C_{\text{n-delay}}^m = \sum_{k=1}^{N_m} d_k^m \lambda_k^m \delta_k^m. \tag{17}$$

Thus, the *total expected end-to-end delay* and *total expected end-to-end delay cost* for all keys from edge server m is

$$D_{\text{delay}}^m = D_{\text{a-delay}}^m + D_{\text{n-delay}}^m, \tag{18}$$

$$C_{\text{delay}}^m = C_{\text{a-delay}}^m + C_{\text{n-delay}}^m. \tag{19}$$

Therefore, the *total expected end-to-end delay* and *total expected end-to-end delay cost* in the whole system is

$$D_{\text{delay}} = \sum_{m=1}^{M} D_{\text{delay}}^m, \quad C_{\text{delay}} = \sum_{m=1}^{M} C_{\text{delay}}^m. \tag{20}$$

PROPOSITION 1. *C_{delay} is increasing in T_k^m for $k \in \mathcal{N}_m$, $m \in \mathcal{M}$.*

This is clear from (15) and (17), and intuitive since larger T_k^m results in more aggregation, which increase the delay.

Average traffic: Traffic is measured as the number of updates sent over the network from edge caches to the center. In our model, a cache miss is proceeded by a update or a update ("flush") follows a cache miss. Hence, the number of updates equals to the number of cache miss. Therefore, the *total expected traffic* caused by all keys from edge server m is

$$TR_{\text{traffic}}^m = \sum_{k=1}^{N_m} \lambda_k^m m_k^m = \sum_{k=1}^{N_m} \frac{\lambda_k^m}{1 + \lambda_k^m T_k^m}, \tag{21}$$

and the *total expected traffic cost* caused by all keys from edge server m is

$$C_{\text{traffic}}^m = \sum_{k=1}^{N_m} c_k^m \lambda_k^m m_k^m = \sum_{k=1}^{N_m} \frac{c_k^m \lambda_k^m}{1 + \lambda_k^m T_k^m}, \tag{22}$$

where c_k^m is a constant characterizing the average traffic (miss) cost for content k.

Therefore, the *total expected traffic* and the *total expected traffic cost* in the whole system is

$$TR_{\text{traffic}} = \sum_{m=1}^{M} TR_{\text{traffic}}^{m}, \quad C_{\text{traffic}} = \sum_{m=1}^{M} C_{\text{traffic}}^{m}. \tag{23}$$

PROPOSITION 2. *$C_{traffic}$ is decreasing in T_k^m for $k \in \mathcal{N}_m$, $m \in \mathcal{M}$.*

This is clear from (22), and intuitive since larger T_k^m results in more aggregation, which decreases the traffic.

Our goal is to find optimal timers $\{T_k^m\}_{k\in\mathcal{N}_m, m\in\mathcal{M}}$ to jointly obtain the optimal tradeoff between *the whole system delay and traffic cost.* To do so, we first define the following functions

$$\begin{aligned} C(\boldsymbol{T}) &= \alpha C_{\text{delay}} + (1-\alpha)C_{\text{traffic}} \\ &= \alpha \sum_{m=1}^{M}\sum_{k=1}^{N_m}\left(\left(\frac{1}{1+\lambda_k^m T_k^m}+1\right)\frac{d_k^m \lambda_k^m T_k^m}{2} + d_k^m \lambda_k^m \delta_k^m\right) + (1-\alpha)\sum_{m=1}^{M}\sum_{k=1}^{N_m}\frac{c_k^m \lambda_k^m}{1+\lambda_k^m T_k^m}. \end{aligned} \tag{24}$$

Proof of theorem 3.1:

PROOF. We first show that $C(\boldsymbol{T})$ is convex in T_k^m for $k \in \mathcal{N}_m$, $m \in \mathcal{M}$. We examine the properties of its second derivative as follows. The first-order derivative of $C(\boldsymbol{T})$ with respect to (w.r.t.) T_k^m is $\frac{\alpha d_k^m \lambda_k^m}{2} + \frac{\alpha d_k^m \lambda_k^m}{2}\frac{1}{(1+\lambda_k^m T_k^m)^2} - \frac{(1-\alpha)c_k^m(\lambda_k^m)^2}{(1+\lambda_k^m T_k^m)^2}$, then its second derivative w.r.t. T_k^m is

$$\begin{aligned} \frac{\partial^2 C(\boldsymbol{T})}{\partial (T_k^m)^2} &= \frac{-\alpha(\lambda_k^m)^2 d_k^m}{(1+\lambda_k^m T_k^m)^3} + \frac{2(1-\alpha)(\lambda_k^m)^3 c_k^m}{(1+\lambda_k^m T_k^m)^3} \\ &= \frac{\alpha(\lambda_k^m)^2 d_k^m}{(1+\lambda_k^m T_k^m)} > 0 \end{aligned}$$

since $\alpha \geq 0$, $T_k^m \geq 0$, $\lambda_k^m \geq 0$ and $d_k^m > 0$. Hence $C(\boldsymbol{T})$ is convex in T_k^m for $k \in \mathcal{N}_m$, $m \in \mathcal{M}$.

Since $C(\boldsymbol{T})$ is convex in T_k^m, for $k \in \mathcal{N}_m$, $m \in \mathcal{M}$, we take its derivative w.r.t. T_k^m and set it to zero, we have

$$\frac{\alpha d_k^m \lambda_k^m}{2} + \frac{\alpha d_k^m \lambda_k^m}{2}\frac{1}{(1+\lambda_k^m T_k^m)^2} - \frac{(1-\alpha)c_k^m(\lambda_k^m)^2}{(1+\lambda_k^m T_k^m)^2} = 0,$$

i.e., $T_k^m = \frac{1}{\lambda_k^m}\left(\sqrt{\frac{2(1-\alpha)c_k^m\lambda_k^m}{\alpha d_k^m} - 1} - 1\right)$. In real system, we require $T_k^m \geq 0$, i.e, $\lambda_k^m \geq \frac{\alpha d_k^m}{(1-\alpha)c_k^m}$. Hence, for all $k \in \mathcal{N}_m$ with $0 \leq \lambda_k^m < \frac{\alpha d_k^m}{(1-\alpha)c_k^m}$, we map its timer T_k^m to zero. □

Derivation of transition points in Figure 3:

PROOF. Without loss of generality, we consider a given key k at a given edge. From Theorem 3.1, we have $T = 0$ if λ is in region I in Figure 3. In the following, we consider the case $\lambda \geq \lambda_0$.

We consider the first derivative of $T(\lambda)$ w.r.t. λ,

$$\begin{aligned} \frac{\partial T}{\partial \lambda} &= -\frac{1}{\lambda^2}\left(\sqrt{\frac{2(1-\alpha)c\lambda}{\alpha d} - 1} - 1\right) + \frac{1}{\lambda}\cdot\frac{1}{2}\frac{1}{\sqrt{\frac{2(1-\alpha)c\lambda}{\alpha d} - 1}}\cdot\frac{2(1-\alpha)c}{\alpha d} \\ &= \frac{1}{\lambda^2}\left[1 - \frac{(1-\alpha)c\lambda - \alpha d}{\alpha d}\sqrt{\frac{\alpha d}{2(1-\alpha)c\lambda - \alpha d}}\right]. \end{aligned} \tag{25}$$

Then, for $\frac{\partial T}{\partial \lambda} > 0$, we have $\frac{(2-\sqrt{2})\alpha d}{(1-\alpha)c} < \lambda < \frac{(2+\sqrt{2})\alpha d}{(1-\alpha)c}$. As $\lambda \geq \frac{\alpha d}{(1-\alpha)c}$, we have $\frac{\alpha d}{(1-\alpha)c} \leq \lambda < \frac{(2+\sqrt{2})\alpha d}{(1-\alpha)c}$. Similarly, $\lambda \geq \frac{(2+\sqrt{2})\alpha d}{(1-\alpha)c}$ when $\frac{\partial T}{\partial \lambda} < 0$. Therefore, T is increasing in λ when $\frac{\alpha d}{(1-\alpha)c} \leq \lambda < \frac{(2+\sqrt{2})\alpha d}{(1-\alpha)c}$, and decreasing in λ when $\lambda \geq \frac{(2+\sqrt{2})\alpha d}{(1-\alpha)c}$. Denote $\lambda_0 \triangleq \frac{\alpha d}{(1-\alpha)c}$, and $\lambda_1 \triangleq \frac{(2+\sqrt{2})\alpha d}{(1-\alpha)c}$.

Next, we consider the second derivative,

$$\begin{aligned}
\frac{\partial^2 T}{\partial \lambda^2} &= -\frac{2}{\lambda^3}\left[1 - \frac{(1-\alpha)c\lambda - \alpha d}{\alpha d}\sqrt{\frac{\alpha d}{2(1-\alpha)c\lambda - \alpha d}}\right] \\
&\quad + \frac{1}{\lambda^2}\left[-\frac{(1-\alpha)c}{\alpha d}\sqrt{\frac{\alpha d}{2(1-\alpha)c\lambda - \alpha d}} - \frac{(1-\alpha)c\lambda - \alpha d}{\alpha d}\cdot\sqrt{\alpha d}\cdot\left(-\frac{1}{2}\right)\frac{2(1-\alpha)c}{[2(1-\alpha)c\lambda - \alpha d]^{3/2}}\right] \\
&= -\frac{2}{\lambda^3} + \frac{[\sqrt{3}(1-\alpha)c\lambda - (\sqrt{3}-1)\alpha d][\sqrt{3}(1-\alpha)c\lambda - (\sqrt{3}+1)\alpha d]}{\lambda^3(\alpha d)^{1/2}[2(1-\alpha)c\lambda - \alpha d]^{3/2}}. \qquad (26)
\end{aligned}$$

Again, consider $\frac{\partial^2 T}{\partial \lambda^2} < 0$, we have

$$[\sqrt{3}(1-\alpha)c\lambda - (\sqrt{3}-1)\alpha d][\sqrt{3}(1-\alpha)c\lambda - (\sqrt{3}+1)\alpha d] < 2(\alpha d)^{1/2}[2(1-\alpha)c\lambda - \alpha d]^{3/2}. \qquad (27)$$

Denote $f_1(\lambda) = [\sqrt{3}(1-\alpha)c\lambda - (\sqrt{3}-1)\alpha d][\sqrt{3}(1-\alpha)c\lambda - (\sqrt{3}+1)\alpha d]$, and $f_2(\lambda) = 2(\alpha d)^{1/2}[2(1-\alpha)c\lambda - \alpha d]^{3/2}$. It is easy to check that both f_1 and f_2 are increasing in λ when $\lambda \geq \frac{\alpha d}{(1-\alpha)c}$, and

$$f_1(\lambda_1) = (8 + 6\sqrt{2})(\alpha d)^2 < f_2(\lambda_1) = (14 + 10\sqrt{2})(\alpha d)^2. \qquad (28)$$

Thus, it is clear that $\frac{\partial^2 T}{\partial \lambda^2} < 0$ when $\lambda_0 \leq \lambda \leq \lambda_1$, i.e., T is concavely increasing in λ when $\lambda_0 \leq \lambda \leq \lambda_1$.

Now, suppose there exists a λ_2 such that $\frac{\partial^2 T}{\partial \lambda^2} = 0$, thus it is clear that $\lambda_2 > \lambda_1$ from (27) and (28).

Therefore, we have T concavely increasing in λ when $\lambda_0 \leq \lambda \leq \lambda_1$, concavely decreasing in λ when $\lambda_1 \leq \lambda \leq \lambda_2$, and convexly decreasing in λ when $\lambda > \lambda_2$.

Since $f_1(\lambda) > f_2(\lambda)$ when λ is large, while $f_1(\lambda) < f_2(\lambda)$ when λ is small, and both f_1 and f_2 are continuous, hence λ_2 indeed exists. We can numerically verify the existence and value of λ_2. □

Proof of corollary 3.2:

PROOF. From (1) and Theorem 3.1, we immediately have

$$\begin{aligned}
B_m &= \sum_{k=1}^{N_m}(1 - m_k) = \sum_{k=1}^{N_m}\frac{\lambda_k T_k}{1 + \lambda_k T_k} = N_m - \sum_{k=1}^{N_m}\frac{1}{1 + \lambda_k T_k} \\
&= N_m - \left(\sum_{k=1}^{k'}\frac{1}{1 + \lambda_k T_k} + \sum_{k=k'+1}^{N_m}\frac{1}{1 + \lambda_k T_k}\right) \\
&= N_m - \left(\sum_{k=1}^{k'}\frac{1}{1 + \lambda_k^m\left(\frac{1}{\lambda_k^m}\left(\sqrt{\frac{2(1-\alpha)c_k^m\lambda_k^m}{\alpha d_k^m}} - 1 - 1\right)\right)} + \sum_{k=k'+1}^{N_m}\frac{1}{1 + \lambda_k^m \cdot 0}\right) \\
&= \sum_{k=1}^{k'}\left(1 - \sqrt{\frac{\alpha d_k^m}{2(1-\alpha)c_k^m\lambda_k^m - \alpha d_k^m}}\right). \qquad (29)
\end{aligned}$$

□

Received February 2019; revised March 2019; accepted April 2019