Low-Latency Analytics on Colossal Data Streams with SummaryStore

Nitin Agrawal
Samsung Research

Ashish Vulimiri
Samsung Research

Abstract

SummaryStore is an approximate time–series store, designed for analytics, capable of storing large volumes of time-series data (~1 petabyte) on a single node; it preserves high degrees of query accuracy and enables near real-time querying at unprecedented cost savings. SummaryStore contributes *time-decayed summaries*, a novel abstraction for summarizing data streams, along with an ingest algorithm to continually merge the summaries for efficient range queries; in conjunction, it returns reliable error estimates alongside the approximate answers, supporting a range of machine learning and analytical workloads. We successfully evaluated SummaryStore using real-world applications for forecasting, outlier detection, and Internet traffic monitoring; it can summarize aggressively with low median errors, 0.1 to 10%, for different workloads. Under range-query microbenchmarks, it stored 1 PB synthetic stream data (1024 1TB streams), on a single node, using roughly 10 TB (100x compaction) with 95%-ile error below 5% and median cold-cache query latency of 1.3s (worst case latency under 70s).

1 Introduction

Continuous generation of time-series data is on a significant rise, particularly from sensors, servers, and personal computing devices [78]; an individual μPMU sensor to monitor electricity consumption generates 50 billion samples per year [18], data centers log hundreds of billions of events per day [61], and each self-driving car is expected to generate several petabytes of data per year [80].

Despite disks becoming cheaper, administered storage remains expensive [59]; the growth in data is far outpacing the growth in capacity and simply adding hardware resources to scale up or out is not cost efficient. Even if one were to keep adding disks for capacity, as datasets grow, analytical tasks become progressively slower. In-memory systems are capable of significantly faster response times but are expensive and do not store data persistently. Time–series stores thus need to meet the competing demands of providing cost-effective storage while maintaining low response times.

Increasingly, algorithms, not human readers, consume time-series data. Many of these algorithmic analyses are near real-time, ranging from data-center monitoring [35, 52, 66], financial forecasting [51], recommendation systems [56, 60], to applications for smart homes and IoT [1, 40, 47, 75, 86]. Significant research in machine learning is devoted to agents that learn on data over extended periods of time [19, 62, 76, 79]. A survey we performed of the various kinds of analyses (§2) offers three major insights into the characteristics of time–series workloads which mandate a fundamental rethinking of time–series storage systems.

First, the analytical tasks explore various aggregate attributes and statistical properties retrospectively for an entire stream, or a sub range, for higher-level applications in forecasting, classification, or trend analysis. Unlike applications using key–value stores and file systems, analytical tasks seldom subject time–series stores to point queries. Second, the vast majority of analyses exhibit a temporal affinity favoring *recent* data over *older*, even when they perform retrospective querying on the entire corpus. Third, in some cases, such as anomaly or outlier detection, certain specific events are tremendously more valuable than others.

To overcome the challenges in building stores for large-scale time-series analysis, we have designed and implemented SummaryStore. Based on the first insight, SummaryStore embraces approximation to answer analytical queries on the data. It maintains compact summaries through aggregates, samples, sketches, and other probabilistic data structures, in lieu of the raw data. Existing summaries are generic enough to serve a wide variety of queries and new ones can be added. SummaryStore preserves an interface similar to conventional time–series stores; applications can 1) insert data and 2) read data by specifying an arbitrary time range. Since it is an approximate store, read responses additionally contain a confidence estimate for the answer.

Authors can be reached at nitina@cs.wisc.edu and ashish@vulimiri.net.

SOSP '17, October 28, 2017, Shanghai, China

ACM ISBN 978-1-4503-5085-3/17/10…$15.00
https://doi.org/10.1145/3132747.3132758

From a scaling viewpoint, uniform approximation leads to an undesirable linear growth in store size with data. SummaryStore uses the second insight to define a novel time-decayed approximation scheme and a data ingest pipeline. Summary construction favors recent data by allocating progressively fewer bytes for older data; essentially the extent of approximation increases with age thereby *decaying* older data. As shown through evaluation, the resulting store provides higher overall accuracy to analyses while compacting the data store sub-linearly or logarithmically.

Finally, the third insight poses a design challenge at odds with approximation; a summarized representation, while significantly advantageous from a storage perspective, alone is insufficient in ensuring that specific events have been captured. SummaryStore treats events of special interest as *landmarks*, stores them in their entirety, and seamlessly combines them with summaries, when appropriate, to answer queries. Read queries automatically get the accuracy benefit if their time range overlaps with landmarks.

By combining decayed summarization with landmarks SummaryStore improves overall accuracy making it feasible to run a broader range of applications, for example, an otherwise infeasible anomaly detection, while reaping the storage benefits of approximation. SummaryStore also opens the door to new machine-learning algorithms by giving practitioners the ability to operate with significantly larger datasets, democratizing their analysis. Further, resource-constrained devices such as smartphones and wearables can locally operate with larger, more sophisticated models.

SummaryStore was evaluated using multiple real-world datasets and applications and demonstrated high-degrees of compaction[1] for small drops in accuracy. In particular, Facebook's Prophet forecasting engine yielded nearly the same forecasts using SummaryStore for 10x compaction on three real-world datasets from economics, climatology, and Internet traffic. Outlier detection for Google's cluster management dataset required about 6x less space using SummaryStore for zero false positives. SummaryStore compacted synthetically-generated *colossal*[2] stream data by a factor of roughly 100x with 95%-ile error below 5%; in particular, stream data totaling 1 PB (1024 1TB streams) was compacted by 100x to about 10 TB (1024 10 GB streams). For cold starts, median latency was 1.3s and worst-case was below 70s; more typical warm-cache latencies are in order of milliseconds.

This paper makes the following contributions which form the basis for the SummaryStore system implementation.

- Compact *time-decayed summaries* for time-series data supporting a range of pragmatic *power-law* [67] decay.
- Novel ingest mechanism to unify summarized and landmark data to accurately answer arbitrary range queries.
- Rigorous statistical techniques to generate accurate responses and error bounds for approximate queries.

Analysis and Apps	Description
BellKor algorithm [56]	Movie recommendation system; winner of Netflix Prize [23]
Contextual bandits [60]	Generate news recommendation
Facebook EdgeRank [53]	Temporal decay of news posts
Twitter Observability [35]	Archive and analyze operational metrics at differential temporal granularity
Etsy Kale [5]	Identify anomalies on recent+old data
Google Prometheus [52]	Monitor time-series data-center logs
Facebook Gorilla [66]	Monitor time-series data-center logs
AT&T Gigamining [34]	Analyze call records in telecom ops
Netflix Edda [10], Atlas [9]	Operational insights+outage analysis
Cohen *et al.* [31]	TCP connections at busy web server
Bremler-Barr *et al.* [27]	Path quality+Internet gateway selection
Smart-home apps [1, 40, 47, 70, 75, 86]	Occupancy sensing, energy monitoring, HVAC control, security
FinTime [41, 51]	Financial time-series analysis
Macrobase [21]	Outlier detection in time-series data

Table 1: Analyses Favoring Recent Data Over Older.

2 Why Build Stores for Temporal Analytics?

Several popular analyses favor recent data over old and query retrospectively on historical data; Table 1 summarizes them.

Recommender systems. These often rely on, and benefit from, temporal information to make recommendations [13, 56, 60]. Koren *et al.*'s BellKor algorithm [56], which won the Netflix Prize [23], makes a distinction between temporal effects, ones that span extended periods of time and ones more recent/transient, for attributes such as movie likeability and user biases; their analysis requires different scales of temporal granularity when modeling different phenomena leading to significant improvements on recommendation accuracy. A detailed survey of the temporal biases of recommender systems can be found elsewhere [29].

Smart-home and IoT apps. These analyze streams of sensor data to make decisions and provide services with the majority sensitive to differences in recent and historical data. One occupancy-sensing app records indicators, such as luminosity, sound, and CO_2 level, over exponentially-increasing time windows for efficiently predicting office occupancy [47]. Another computes both long- and short-term statistics over sound samples for people-counting in order to control the HVAC [86]. Similar apps for energy monitoring [1] compute historical area-under-the-curve for energy meters and temporal energy usage [75]. An app for controlling the thermostat faces an instability in actuation if only a small time window is considered [40]; an app for detecting physical activity using accelerometers [70] overcomes the same challenge by accounting for both recent and historical movement data.

[1] For raw stream data of size S, and SummaryStore's decayed size s, compaction is defined as the factor S/s.

[2] An informal reference to stream data with terabytes to petabytes of unsummarized size by virtue of containing billions or trillions of events.

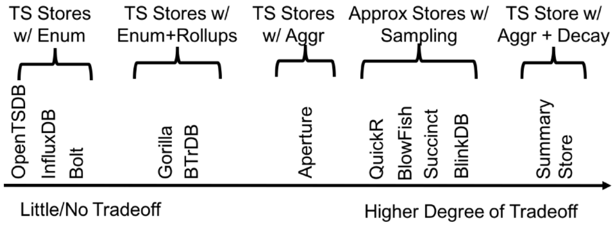


Figure 1: Stores providing storage vs accuracy tradeoff.

TS Store	Size (GB)	Cost (USD)		Time of Range Query: Secs (Error)		
		HDD	SSD	Scan	Large	Small
Influx	196	$10	$118	1347 (0)	1263 (0)	27 (0)
SStore						
100x	2	$0.1	$1.2	13 (0)	11 (1%)	3 (2%)
10x	20	$1	$12	112 (0)	101 (.1%)	28 (.1%)

Table 2: Cost and runtime for time-series stores. Comparison for stream w/ 10 billion events over a year with 20 byte values. Latency for 3 range-count queries: scan, large range (80% of full), small range (random 2% of full). Query error in parenthesis (InfluxDB is always 0). Cost calculated as fractional price for 512GB SSDs ($0.60/GB) and 4TB HDDs ($0.05/GB) [48].

DevOps and monitoring. Tools for data visualization and dashboards are commonly deployed across services and enterprises to monitor operational time-series metrics [6]. Etsy's Kale identifies anomalies over recent data and subsequently searches through historical data to find similar anomalies [5]. Twitter's Observability archives operational metrics at a lower granularity, for trending and long-term analysis, whereas higher granularity, performance critical, data is periodically expired [35]. Sysadmins pose queries over a variety of activity, recent and historical, including over data backups in IBM TSM [11], monitoring time-series data at Google [52] and Facebook [66], and operational insights [9] and outage analysis [10] at Netflix.

Financial analyses. These are used for forecasting and identifying trends [41]. For example, a financial time-series benchmark, FinTime [51], lays out the typical queries over historical market information and ticks for financial instruments; it consists of a combination of deep historic queries, short time-depth queries, and time-moving statistics.

Aren't existing stores adequate for time-series data? Figure 1 organizes popular time-series stores per their flexibility in trading-off space for accuracy; an in-depth comparison is presented in §8. Providing a high degree of tradeoff poses systems and algorithmic challenges. Conventional time–series stores *enumerate* all raw data (enum stores for short), and a few maintain additional aggregations to reduce query latency, further increasing the cost of data storage; both these approaches are non-trivial to scale. Approximate stores provide acceptable, albeit imperfect, answers quickly and at low cost through algorithmic constructs. However, existing approximate stores are not designed for time-series data; furthermore, they rely primarily on sampling, and provide no guarantees for storing specific events of interest, rendering them unviable for most time-series analytics.

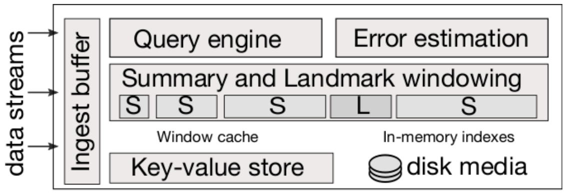


Figure 2: SummaryStore system architecture. S, L refers to summarized and landmark windows.

Table 2 illustrates the inadequacy of conventional time–series stores, and the potential benefits of careful approximation, using a simple experiment. A synthetically generated time-series of 10 billion events is inserted into InfluxDB [50], a popular open–source time–series store, and SummaryStore. The table lists the resulting store size, cost, and latency for typical range queries, along with the query accuracy (InfluxDB is always 100% correct). Even on this relatively modest dataset, InfluxDB's storage cost and query latency are significantly higher compared to SummaryStore. As datasets grow, the differences are bound to be starker, further motivating the need for a new time–series store.

3 Overview of SummaryStore

SummaryStore is built for analytical and machine-learning workloads which extract insight by analyzing trends, patterns, and other statistical attributes. SummaryStore thus ingests time-series data and supports temporal range queries by maintaining a careful summarization that supports the analyses. To aggressively compact data while being able to accurately answer a variety of queries, SummaryStore relies on the insight that most analyses favor recent data over older, even when they require retrospective analysis over historical data. SummaryStore thus approximates and *decays* the majority of data so that the relative contribution of each data item to the store is scaled down proportional to its age [32]; time-decayed summaries (TDS) are its novel construct to represent a data stream[3]. Figure 2 shows the system architecture.

SummaryStore represents a time series as *windows* over contiguous spans of time; post ingest, most data is subsumed in *summarized* windows, a carefully approximated digest in lieu of raw data, with comparable capability to answer certain types of queries; a relatively smaller subset of specific entries are stored in full in *landmark* windows.

[3] We refer to time-series data as data streams and time-series analytics as temporal analytics interchangeably.

`CreateStream(decay, [Summary Operators])`	Create new stream with specified decay parameters and, optionally, a set of summary operators for this stream; default set is all available operators.
`DeleteStream(stream)`	Delete the specified stream in entirety.
`Append(stream, [timestamp], value[])`	Add time-value data to stream; `value` can be list of attribute–value pairs; timestamp optional.
`BeginLandmark(stream)`	Start new landmark window effective *now*.
`EndLandmark(stream)`	End current landmark window effective *now*.
Q_s: `Query(stream,` T_s`,` T_e`, operator, params)`	Query stream over specified time range using specified summary operator; some queries warrant additional parameters (*e.g.*, `value` when checking membership using Bloom filter).
Q_l: `QueryLandmark(stream,` T_s`,` T_e`)`	Query stream over specified time range for landmark time-value entries, if any.
`Response` Q_s`: (answer, confidence estimate)`	Query response when over at least one summarized window: likely answer and error estimate.
`Response` Q_l`: {[timestamp], value[]}`	Query response over landmark windows only: an enumeration of time-value pairs.

Table 3: SummaryStore API.

3.1 Summary operators

Choice of operators: SummaryStore employs multiple sets of summary operators. The first set records simple aggregates Count, Sum, Mean, and Min/Max. The second set allows frequency estimation and counting through Histogram, Quantile, Count-min sketch (CMS), Counting Bloom filter, and Hyperloglog counter. The third set answers membership queries using a Bloom filter. The fourth set allows arbitrary queries through a Sampled subset of data. Each chosen set is well suited to answer specific types of queries. For example, a sampled summary is highly effective for SQL-style selections and projections, in contrast to aggregates, but inefficient for simple Counts and Sums; for the latter, aggregates can be additionally maintained for a low overhead.

Configuring summaries: Each stream can be independently configured to use specific summary operators. When available, a-priori knowledge of types of queries can be used to configure a stream with only the most relevant sets to save space. In the absence of such information, the default is to use the entire collection which, typically, tends to still be substantially smaller than the raw data.

Adding new summary operators: New operators can be added to SummaryStore as long as they specify a *union* function, *i.e.*, two instances of the same type of operator can be unioned [14] to produce another instance of the same operator. For example, the union of two Counts is addition; the union of two Bloom filters, with the same size and hash functions, is a bitwise OR; for sampling, two windows with N samples each are re-sampled to a single one with N. The implication of unioning is discussed in §4; the operators chosen thus far satisfy this requirement.

3.2 Data decay

Summarized windows span progressively-longer time lengths with age, *i.e.*, *older* windows span larger times compared to *recent.* Since windows are allocated a constant storage budget, data decays as it ages by being subsumed in larger-time-span windows (representing more raw data). For example, each window's Bloom filter [26] is configured to the same hash functions and bit-array size; older Bloom filters represent more values, progressively reducing data's representation for membership queries. Streams can be configured from a family of decay functions with different compactions.

To store specific data at full granularity, as summaries may inadvertently smoothen out events of interest, at ingest SummaryStore allows annotating certain intervals of time as *landmarks*; these windows are unaffected by the summarization process and are stored in full. SummaryStore carefully aligns landmark and summarized windows to support queries seamlessly over the entire data stream, weaving them together into one contiguous time span (§4).

Windows are useful logically, in organizing a data stream as temporal segments to provide bounds on query errors (§5), and physically, in managing disk I/O. Since windows are internal to SummaryStore, the application interface remains similar to other time–series stores.

3.3 SummaryStore API

SummaryStore has a simple, yet powerful, interface to write and query streams as shown in Table 3. For appends, if a timestamp is not specified, one is assigned using the system clock. To prune old data, the store relies on decay instead of explicit deletes. We give a few query examples:

What was the avg. energy consumption last month?
`Query`(energy_readings, now() - 1 mo, now(), average)
Did a particular node back up last week?
`Query`(backup_log, now() - 7d, now(), existence, nodeID)
How many times did a user visit the server in 2015?
`Query`(visitors, 2015-01-01, 2015-12-31, freq, client IP)

Since summaries are maintained at a window granularity, SummaryStore must handle the necessary imprecision for queries not window-aligned. SummaryStore uses statistical modeling to return (i) its maximum likelihood answer to the query, and (ii) a reliable error estimate as a measure of the operator-specific uncertainty; we discuss details in §5.

3.4 Limitations

The current SummaryStore prototype suffers from certain limitations which can form the basis for future work.

Algorithm 1: Merge algorithm for time-decayed ingest

```
configured : D = decay function, expressed as a seq. of window lengths
             (e.g. 1, 2, 4, 8, . . . )
function append(timestamp, value)
    W ← new summary window
    if in landmark then
        L ← current landmark window
        L.append (timestamp, value)
    else
        W.update (timestamp, value)
    summary_windows.add (W)
    foreach consecutive summary windows W1, W2 do
        if $\exists K \mid \sum_{i=0}^{K} D[i] \leq W1.start$ and $W2.end < \sum_{i=0}^{K+1} D[i]$ then
            /* i.e. if W1 and W2 are completely contained
               inside Kth target window                        */
            merge(W1, W2)
```

- The set of chosen summary operators can answer a variety of queries but are nonetheless restrictive in the high-level analytics they support; new operators can be added to broaden the scope but come with challenges. Each new operator requires an error estimator similar to the ones in §5. Also, not all statistics are unionable, *e.g.*, median.
- SummaryStore can be configured to yield a desired compaction (§4.2) but the consequent impact on the accuracy of individual queries is hard to determine a priori. While higher compaction generally leads to lower accuracy, the specific extent can be unpredictable. Different applications can have different tolerance even to the same errors.
- Landmarks provide crucial support for handling unsummarized data but their identification at the time of data ingest can be a challenge (§4.3). Landmark criteria need to be predefined and face limitations conceptually similar to other approximate systems; *e.g.*, stratified sampling requires stratum weights configured a-priori as well [3].

4 Time-Decayed Data Processing (Appends)

SummaryStore needs a mechanism to continually ingest and decay data; for this it builds on algorithmic work on time-decayed aggregates [32, 33, 39]. Prior work has shown how to construct an exponential windowing to approximately track arbitrary time-weighted counts in integer streams using $\Theta(\log^2 N)$ bits, as opposed to the $\Omega(N)$ needed for an exact count [32, 39]. While aggressive exponential decay suffices for this specific problem, we find that general workloads benefit from a gentler, more controlled, rate of decay (§7). Further, SummaryStore must also maintain portions of the data in landmark windows at full resolution. SummaryStore's time-decayed summaries and data ingest mechanism address these challenges by 1) supporting a broad range of gentle *power-law* [67] decay functions which are more practical compared to exponential; 2) integrating landmark and summarized windows, carefully aligning them at ingest to ensure queries perceive a seamless, unified view of the data.

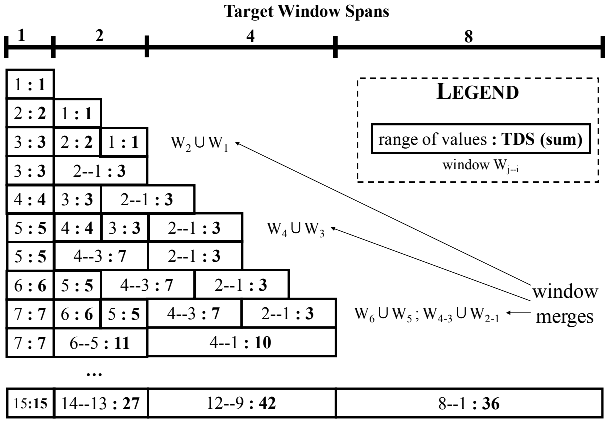


Figure 3: Time-Decayed Merge Algorithm Sample Run. Evolution of windows using merge algorithm as the stream of numbers 1,2,3,4,... is written to SummaryStore configured with exponential windowing [1,2,4,8,..]. Merge operations ($\cup$) listed as they occur. For this decay, there will be $\Theta(\log N)$ windows after N inserts.

4.1 Window merge algorithm for time-decayed ingest

At the core of SummaryStore's data ingest is an online procedure that continually recompacts older data, as it ages, to decrease its granularity, when new data is added to the stream; this is essential in maintaining time-decayed aggregates on a stream since the continuous addition of data forces a continual evolution of window boundaries. The key primitive is a *merge* operation that combines a pair of consecutive windows into one, effectively halving the storage budget allocated to the values spanned in the two windows.

Algorithm 1 details SummaryStore's ingest and merge mechanism. The algorithm calls less than one amortized merge operation for every element ingested: the precise cost depends on the chosen rate of decay, with less aggressive decay requiring less frequent compaction. In practice, we batch appends to further reduce processing cost.

Figure 3 shows a simple example for ingest of the stream of numbers 1,2,3,4,... using the underlying merge algorithm. The SummaryStore instance is configured with target window sizes [1,2,4,8,16,..]. Note that these target sizes happen to be exponential for illustration; in practice, the target window sizes are defined by the specific power-lay or exponential decay function, and can take on any sequence. As an example, PowerLaw(1,1,1,1) will define target sizes [1,2,3,4,..]. For simplicity, the time-decayed summary (TDS) in this example is configured to only maintain the Sum of values per window. The union $\cup$ for Sums is simply their addition. The first three inserts (of values 1, 2, 3) create three Summary windows W_1, W_2 and W_3 with respective Sums 1, 2 and 3. After the third insert, since W_1 and W_2 are aligned within the

Decay function	Parameters	Size of store after n appends	Decay rate (# bits to nth oldest element)	Sequence of window lengths	Example storage growth
Power law	$p, q, R, S \in \mathbb{N}$	$\Theta\left(R\left(\frac{n}{RS}\right)^{\frac{p}{p+q}}\right)$	$\Theta\left(\frac{1}{S}\frac{p}{p+q}\left(\frac{n}{RS}\right)^{\frac{-q}{p+q}}\right)$	for $k = 1, 2, .. : R\ k^{p-1}$ of length $S\ k^q$	PowerLaw(1,1,10,1) = $\sqrt{\frac{N}{10}}$
Exponential [39]	$b \in \mathbb{R}, R, S \in \mathbb{N}$	$\Theta\left(R\log_b \frac{n}{RS}\right)$	$\Theta\left(\frac{1}{S\log b}\frac{1}{n}\right)$	for $k = 1, 2, .. : R$ of length $S\ b^k$	Exponential(3,1,5) = $5\log_3 N$

Table 4: Decay functions in SummaryStore. Family of Power Law and Exponential decays configurable in SummaryStore. Appendix A provides detailed description and derivation of these results.

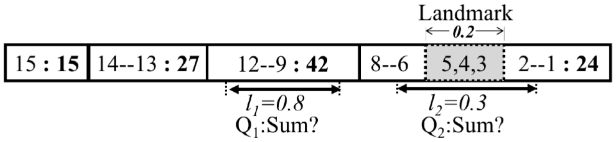


Figure 4: Challenge in answering sub-window queries for Summarized and Landmark windows. Q_1 asks Sum over summarized only, Q_2 over summarized and landmark. For the first window, 24 refers to the sum for 8—6 and 2—1; {3,4,5} are excluded from summaries and included in the landmark window.

	Individual stream size		
PowerLaw (p, q, R, S)	*10 GB*	*100 GB*	*1000 GB*
(1, 1, 88, 1)	1.1x	3.4x	11x
(1, 1, 16, 1)	2.5x	7.9x	25x
(1, 1, 8, 1)	3.5x	11x	35x
(1, 1, 4, 1)	5x	16x	50x
(1, 1, 1, 1)	10x	32x	100x
(1, 2, 48, 1)	22x	100x	480x
(1, 2, 5, 1)	100x	460x	2200x
Exponential (b, R, S)	*10 GB*	*100 GB*	*1000 GB*
(2, 88, 1)	120x	1100x	9700x
(2, 32, 1)	320x	2800x	25000x
(2, 1, 1)	8600x	77000x	700000x
(3, 1, 1)	14000x	120000x	1100000x

Table 5: Storage compaction evolution w/ decay configurations. Column name = size of raw data, increasing over time; compaction = (size of raw data)/(size of SummaryStore). The parameters of the power-law decay function map to different window lengths and consequently different compactions; admins can refer to table as rule-of-thumb for configuring.

same target window boundary (of length 2), they are merged into a single window W_{2-1} with Sum 3. A similar merge results after another two inserts (4, 5), leading to the windows W_5, W_{4-3} and W_{2-1} with Sums 5, 7 and 3. After the next two inserts (6, 7) W_{2-1} and W_{4-3} are in turn merged, as are W_5 and W_6, to result in W_7, W_{6-5} and W_{4-1} with Sums 7, 11 and 10. Each window contains a single Sum over progressively-increasing numbers of values tracking the desired decay. This wave-like process repeats itself on each insert; in steady-state, window merges are amortized and, for exponential, leads to $\Theta(\log N)$ windows after N inserts.

After a merge, the new window inherits the start time of the earlier window, the end-time of the later window, and a union of the Sums in the two windows. The TDS typically maintains a richer set of data structures and the merge entails the union of all the operators.

4.2 Understanding and configuring decay functions

The *decay function* governs the rate of decay of a time series and manifests as the sequence of window lengths. SummaryStore introduces gentler power-law decay functions, in addition to exponential, allowing storage growth rates of $\Theta(n^{\frac{p}{p+q}})$, as listed in Table 4. The parameters of the power-law decay function p, q, R, S dictate the sequence of window lengths and the decay rate to control the overall compaction for a stream. Table 5 shows the growth in storage footprint as a stream of 16-byte tuples is written to the store, for various configurations of the power-law parameters.

Parameters p and q control the overall rate of growth. For instance, with $(p, q) = (1, 1)$ storage use grows as $O(\sqrt{N})$; thus in the first five rows of Table 5, as the amount of raw data ingested grows 100x from 10 GB to 1000 GB, the size of the store increases by $\sqrt{100}$x = 10x. Note that growth is always *sublinear* with power-law decay: storage use always grows more gently than the rate of data arrival.

R and S throttle growth by constant factors. Essentially, $R * k^{(p-1)}$ windows are created for lengths $S * k^q$, with monotonically increasing k. For example, with PowerLaw(1,1,1,1), exactly 1 window each for lengths 1,2,3,..., will be created compared to 16 for PowerLaw(1,1,16,1), and overall compaction with the latter decay is always $\sqrt{16}$ = 4x lower, as Table 5 shows. The larger the value R, the more staggered the growth of windows, and the gentler the decay. Table 5 serves as a rule-of-thumb reference for configuring SummaryStore compaction.

Unlike the broad configuration space enabled by power-law decay, with incremental factors of growth, exponential decay always compacts aggressively. Setting large values for the throttling factors R and S can barely contain the exponential growth in small streams, *e.g.*, Exponential(2,88,1) in Table 5; as data volumes increase, window sizes grow rapidly, and eventually the size of the store almost becomes a constant, growing as $O(\log N)$, as Table 5 shows.

4.3 Landmark Windows

The previous example shows window construction and merge for SummaryStore with only summarized windows; the algorithm can also handle the additional challenge of merging landmarks. A landmark window stores in full resolution any data that lies within its time span. Of course, the larger the extent of the landmark, the more storage space it consumes.

SummaryStore provides the *mechanisms* for applying landmark annotations (*begin* and *end*) but leaves the *policy* decision of identification to the application; we've tested with simple ones, such as the Three Sigma rule (value > 3σ [4]), that often suffice but are by no means exhaustive. The sysadmin or the data scientist using SummaryStore can apply more sophisticated domain-specific rules as needed.

Figure 4 revisits the example in Figure 3 with values {3,4,5} as landmarks instead. W_{8-1} still spans the same time period, but its TDS:Sum only represents the Sum of the values 1..2 and 6..8, maintaining the three values {3,4,5} separately in full. Essentially, every landmark window "hollows out" the time span from each Summarized window it overlaps. The window merge algorithm only merges Summarized windows and leaves landmark windows intact. As we discuss in §5, this does not adversely affect the accuracy of any query; for instance, a Sum over W_{8-1}'s time span will use the TDS:Sum and the Sum of the contained landmarks yielding the same answer (36) as before. In addition, a query seeking specific values within LM_1's time span will get a precise answer.

5 Answering Temporal Queries (Reads)

As an approximate store, SummaryStore needs to provide 1) an accurate likely answer, and 2) a reliable quantification of error. For many queries, in the absence of a precise answer, the store must supply its most likely estimate, as shown in Figure 4; given that it only knows the SUM for an entire window, what should SummaryStore return for the sub-window queries Q_1 and Q_2? One option is storage-intensive models such as an autoregressive moving average (ARMA) [83] which defeat the purpose of summarization; SummaryStore instead employs storage-efficient statistical heuristics.

5.1 Providing likely answers

For count-oriented queries, *e.g.*, Count, Sum and Frequency, in the absence of any additional per-window metadata, the answer proportional to the normalized sub-window length l turns out to be the maximum likelihood answer. The proof (Appendix B) follows from the linearity of counts and the fact that since the arrival process cannot localize inside windows, two queries within the same window with identical sub-window durations must have the same answer.

For summarized windows alone, the proportional length provides the best estimate; in Q_1, $0.8 * 42 = 33.6$. In the presence of landmarks, the best response is the cumulative of summarized windows, proportional to the query overlap excluding any landmarks, along with precise enumeration in the landmarks; in Q_2, $(0.3 - 0.2) * 24 + (3 + 4 + 5) = 14.4$.

For membership-oriented queries the response remains the same as the full window, it being infeasible to answer proportionally. The confidence estimate reflects the increased error uncertainty (§5.2).

5.2 Providing reliable confidence estimates

Using window summaries alone, SummaryStore can provide large and potentially less useful upper bounds for the error, while maintaining extra metadata per-window can be memory intensive. Instead, SummaryStore uses novel statistical methods developed to construct reliable confidence estimates. The methods are frugal with storage; SummaryStore tracks only four values over the entire stream – mean and variance in interarrival times, and mean and variance over values in numeric streams – to estimate query errors. SummaryStore does not rely on individual tests for distributional fit since the general cases subsume the specific ones; this also ensures that the stream model is both compact and independent of the size of the stream. Table 6 lists the methods; proofs are presented in Appendix B. We illustrate through two example operators, Count and Bloom Filter, which require different statistical constructions.

Count: a default response is the upper-bound 100% confidence interval [0,C], which can be too wide to be of practical use. If the stream is known to be Poisson, a significantly tighter estimate is possible: the memorylessness of the Poisson process means each arrival is independent and uniformly at random in $[T_1, T_2]$, thus the distribution of arrivals in a sub-window $[t_1, t_2]$ has a simple Binomial shape. For general `i.i.d.` interarrivals, a closed-form analysis turns out to be much more difficult. However, for large window sizes, where in fact reliable error estimates matter the most, renewal-theoretic techniques [36] can be used to approximate the arrivals by a normal distribution (Appendix B).

Membership (using Bloom filter): If we have a single Bloom filter over the values in $[T_1, T_2]$, how do we use it to answer membership queries on the sub-range $[t_1, t_2]$? An additional challenge here is that unlike counters, Bloom filters are themselves probabilistic data structures which can return false positives [26]. Our error analysis must thus reason jointly about this data structure error and the error due to sub-window interpolation.

Precisely, if the Bloom filter returns `false` for $[T_1, T_2]$ (with 100% certainty [26]), the answer remains the same for $[t_1, t_2]$. If it returns `true`, we need to compute an updated false-positive rate. While it is possible to define a false-positive rescaling with a simple Bloom filter, a more precise answer is possible if we use a frequency data structure such as a *counting* Bloom filter (useful summary on its own too).

Query	Method for Error Estimation
count[a, a+t] (generic)	$N\left(C\frac{t}{T}, \left(\frac{\sigma_t}{\mu_t}\right)^2 \frac{T}{\mu_t}\frac{t}{T}\left(1-\frac{t}{T}\right)\right)$
count[a, a+t] (Poisson)	Binomial $\left(C, \frac{t}{T}\right)$
sum[a, a+t]	$N\left(S\frac{t}{T}, \left(\frac{\sigma_t^2}{\mu_t^2}+\frac{\sigma_v^2}{\mu_v^2}\right)\frac{T\mu_v^2}{\mu_t}\frac{t}{T}\left(1-\frac{t}{T}\right)\right)$
membership(v)[a, a+t]	For Bloom filter with FP probability p: $p\frac{t}{T}$
membership(v)[a, a+t]	For CMS: Pr (Hypergeom(C, S, V) > 0)
frequency(v)[a, a+t]	Hypergeom(C, S, V)
S = normal distribution of count[a, a+t] (generic) in first row V = distribution over frequency(v)[entire window]; v refers to values	

Table 6: Statistical methods for sub-window queries. Error estimation methods developed for sub-window queries using stream modeling. SummaryStore maintains interarrival mean, stdev (μ_t, σ_t), and value mean, stdev (μ_v, σ_v) for an entire stream, not per window, which are used in these methods.

For a Poisson arrival process, the probability of the answer lying in the subrange can be modeled after a Binomial distribution with N occurrences of value V per the frequency summary. For `i.i.d.` arrivals, we rely on the normal distribution for Counts to model the probability distribution as Hypergeometric (Appendix B).

6 Implementation

SummaryStore is implemented in about 7500 lines of Java code. The implementation supports all the operators and decay functions from §4 and exposes an extensible API to allow new decay functions and summary data structures.

Currently SummaryStore uses RocksDB [72] as the storage backend, augmented with an in-memory cache to speed up window accesses; the choice was made primarily for its good append performance and is not tied to the architecture (Figure 2). Windows are stored as objects keyed by a unique ID. We assign IDs to lay out windows (i) grouped by stream, and (ii) sorted in temporal order within each stream, in each level of RocksDB's LSM tree. Each window consists of a configurable number of summary data structures. We augment RocksDB with two in-memory indexes to improve performance: a tree mapping time ranges to windows, used when processing queries, and an efficient heap used by the merge procedure (Algorithm 1) to identify candidate window merges when ingesting new data.

Data is buffered on ingest and processed into the store in batches. SummaryStore serializes all window objects using ProtoBuf [2] which simplifies the on-disk representation across data structures, for example, a variable-length bit array in Bloom filters. SummaryStore uses the Apache Commons Math library [20] for statistical calculations (such as inverting Normal and Hypergeometric distributions) needed when computing error estimates.

7 Evaluation

These questions motivate SummaryStore's evaluation:

- Can it run compelling real-world applications? (§7.1)
- What is insert and query performance for "colossal" stream data? Can querying be done accurately at scale? (§7.2)
- Do error estimates improve at lower compaction? (§7.3)
- How does it perform for various application traces? (§7.4)

The default setup is a server with two 8-core 2.5GHz Intel Xeon CPUs, 64GB RAM, and 5400rpm 500GB disks. The on-disk size ("du -s") benefits from ProtoBuf and RocksDB compression. To ensure comparability, we exclude this in the paper, and report compaction as measured post ingest and prior to disk write.

7.1 Real-world applications

We evaluate SummaryStore's effectiveness in running complex, real-world applications; we chose forecasting and outlier detection as they represent two challenging workloads with very different sets of requirements.

7.1.1 Forecasting on time-series data

Workload. Many applications (§2) use forecasting techniques to predict future values based on the past. We use Facebook's Prophet [77], an open-source engine for forecasting time-series data built on the popular Stan language/ library for maximum likelihood estimation [28]. Prophet is reportedly employed in a variety of applications across Facebook [68], including capacity planning, workload forecasting, and decision-making. We partitioned each time-series from several chosen datasets into train and test data, and configured Prophet with different SummaryStore instances holding training data as full enumeration (no decay or sampling), SummaryStore with uniformly-sampled data, and SummaryStore with time-decayed samples (both power-law and exponential). The uniform-sample store represents the baseline approximate store; intuitively, if this store matches or outperforms time-decayed summaries, decay is not helpful and uniform approximation is sufficient.

We issued Prophet forecast queries for each time-series in each dataset and compared accuracy against test data. The tests for both uniform and decayed stores were run for various levels of storage compaction.

Dataset. We run unmodified Prophet to make forecasts on three varied, real world, time-series datasets: Wikipedia traffic[84], NOAA GSOD weather[64], and Federal Reserve economic[45]. The Wiki dataset contains per-page hourly traffic statistics for Wikipedia articles going back to 2007 (~5TB). The NOAA dataset (~800GB) contains global surface weather data from the USAF Climatology Center collected daily from over 9000 stations between 1929 and 2016. The Econ dataset has 420K time series from 81 sources covering a variety of fiscal and economic indicators such as consumer price indexes, employment and population.

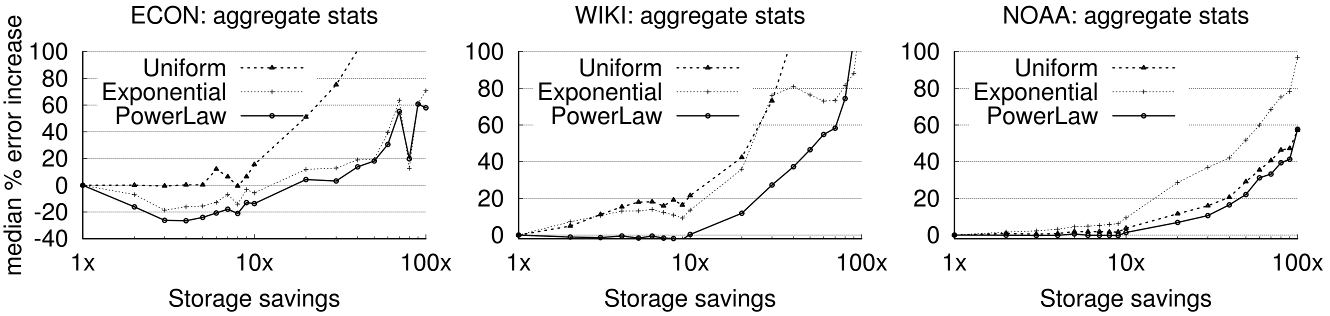


Figure 5: Forecasting accuracy for Facebook Prophet using SummaryStore. All three lines are SummaryStore instances configured to consume the same total storage but with different approximation (uniform sampling w/ no decay, exponential decay, power-law decay). y-axis is median increase in forecast error with 1x being the baseline achieved with all of raw data. x-axis is storage compaction for three datasets: Econ, Wiki, and NOAA; SummaryStore with power-law decay outperforms uniform and exponential for all three.

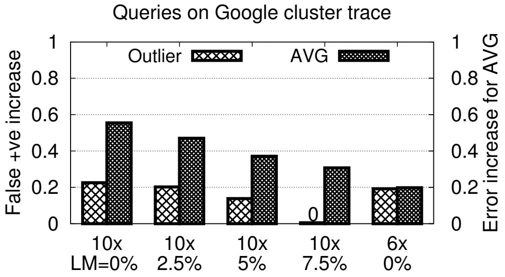


Figure 6: Outlier Detection. With 10x summaries and 7.5% landmarks (net 6*x*), false +ve is 0%; summary-only 6*x* has 19% false +ve and slightly better AVG.

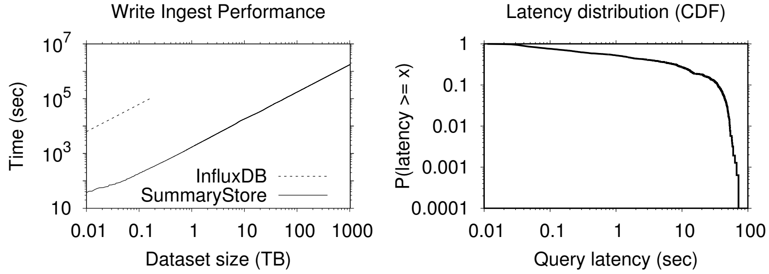


Figure 7: (a) Append performance and (b) query latency. Shown as time taken to ingest billions of time-value pairs compared to InfluxDB. Latency CDF shows distribution across all (age, length) classes.

Results. SummaryStore instances with Uniform, Exponential (b=2), and PowerLaw decay (p=1, q=9) are evaluated; parameters R and S are varied to change the level of compaction for both exponential and power-law. We partition each time series using the last 10% as test data and the remainder as train data. Figure 5 shows the median percentage increase in forecast error relative to the baseline with the unsummarized dataset; there are three takeaways.

First, SummaryStore is remarkably well suited for real-world forecasting under diverse workloads; the difference in accuracy is imperceptible at significant levels of compaction, making it a viable storage system. At 10x compaction, the error is barely noticeable ($< 1\%$); in fact, for Econ, there is a net improvement of ~15% due to decay diminishing the effect of older outliers in the stream. For the other datasets, the error remains $< 10\%$ for $20 - 30x$ compaction. These are significant gains already; if the Prophet algorithm were to be tuned to leverage time-decay, we believe there are further gains to be had. Second, power-law decay as introduced by SummaryStore outperforms exponential, across the board and, by substantial margins in two of the three datasets ($30 - 40\%$ in NOAA and Wiki). Exponential decay heavily favors the most recent observations; while this works to its advantage in Econ, by helping avoid older outliers in the stream, in Wiki and NOAA this loss of historical information leads to a significant reduction in accuracy. Third, power-law significantly outperforms uniform, about 20-30% better for Econ and Wiki. For NOAA, uniform and power-law perform comparably. The reason lies in the regularity of the NOAA dataset, a daily list of temperatures over 100 years; the regularity of the dataset makes it highly predictable for both.

7.1.2 Outlier detection

Workload. Outlier and anomaly detection are key applications in time-series analytics. While approximation poses a challenge to outlier detection, we show that SummaryStore can leverage landmarks to effectively support such analyses at significant cost savings, thus showing that the store can cater to a highly diverse set of analytical applications.

We run an outlier detection workload a commercial analytics service [58] uses to process operational data from its customers; the workload divides time into a number of small intervals and runs a standard boxplot statistical test [63, 65] on each interval to check for the presence of outliers.

Dataset. We use CPU utilization logs from Google's cluster dataset [71, 85] with 1.2 billion observations over 29 days.

Results. The baseline result is a list of time intervals identified as containing outliers using an enum store. Figure 6 shows the relative increase in false positives using SummaryStore. The dataset is particularly outlier-heavy with 60% of windows containing at least one outlier making it difficult to summarize. On the x-axis, the first bar shows that at 10x compaction, and no landmarks, SummaryStore yields roughly 23% more FPs (no false negatives). With an additional 2.5% storage for landmarks, FP rate drops to 20%, and with 5% drops further to 14%. At 7.5% landmarks, the store is able to maintain all the anomaly events and the rate drops to 0%; an effective compaction of roughly 6x. As a sanity check, if the space were instead given to summarized windows for a net 6x compaction, the FP rate merely drops to 19%.

As a contrast, we also ran a query to compute *moving averages*, a typical aggregation query, for which, as expected, error reduces with decreasing compaction; more so when the space is given to summarized windows instead. However, for the same storage as the 6x summary-only config, the 10x w/ 7.5% LM performs only slightly worse ($<$ 10%) for the averages while being significantly better for outliers, providing an operating sweet spot. SummaryStore is thus effective, and efficient, for certain specialized workloads at a modest storage increase, alongside aggregation workloads.

7.2 Scaling to "colossal" data streams

The goal here is to evaluate a SummaryStore instance containing 1 PB worth of stream data, on a single node, in terms of performance (throughput, latency) and query accuracy.

7.2.1 Performance

Write Throughput. These experiments were run on the default server but with 224GB DRAM and a software RAID0 (striped) over 12 1TB disks. Figure 7(a) shows ingest performance for 1 PB of synthetic stream data consisting of 1024 1TB streams each with 62.5 billion 16 byte time-value pairs (8-byte timestamp, 8-byte value). SummaryStore was configured to decay with PowerLaw(1,1,1,1) for 100x compaction leading to a summarized store of ~10 TB. For memory efficiency, the data was ingested in batches of 8 streams allowing the entire summarized working set to stay memory-bound, yielding an ingest throughput of roughly 50 TB/day or 36 million inserts/sec. Disk-bound SummaryStore ingest is ~10x slower. In comparison InfluxDB took about 27 hours for 10 billion inserts (~200 GB) at a steady 100K inserts/sec.

Query Latency. Figure 7(b) shows the distribution of response times for a total of 16000 queries over all time ranges described in §7.2.2; each query asks for the Count in a time range on a random stream and results are computed for 1000 trials in each range. All measurements are with a cold cache: we drop the in-memory SummaryStore, RocksDB and (kernel) page caches before every single query. In practice, queries would typically benefit from caching so these are worst-case bounds. We find that latencies are stable and low, with median 1.3s and worst-case under 70s. SummaryStore outperforms InfluxDB by orders of magnitude; SummaryStore's 1PB instance has worst-case latency <70s compared to 1200s for a 200GB InfluxDB instance (Table 2).

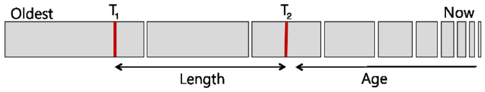


Figure 8: Defining Query Age and Length.

7.2.2 Accuracy, performance for microbenchmarks

Due to temporal decay, two attributes affect query outcomes, namely *age* and *length*; Figure 8 defines these for a query over $[T_1, T_2]$. We first describe the metrics in this evaluation – error, confidence intervals, and latency – followed by the workload, datasets, and results.

- Query Error: With decay query error is expected to be worse over older data relative to recent. *Error is expected to increase with age.* For query length, error is dictated by the extent of the overlap with aligned windows. As discussed in §5, while summaries are maintained at a window granularity, queries don't need to be. *Error is generally expected to decrease with length, with some exceptions.*
- Confidence Intervals (CIs): For a given precision, *e.g.*, 95%-ile, we measure CI widths relative to the baseline answer; a smaller width indicates a more precise, and useful, error bound. SummaryStore's confidence in the answer follows the same trend as the error. *CI width is expected to increase with age and generally decrease with length.*
- Query Latency: Summarization reduces the amount of data to be read for a query lowering latency overall. Since older data is more summarized, *latency is expected to decrease with age.* Similarly, with more data read, *latency is expected to increase with length.*

Workload. We organize time into 4 temporal ranges of the order of minutes, hours, days, and months for a synthetic data stream spanning an year; a given query is characterized by independently selecting *both* an age and a length, for a total of 16 (age, length) classes. We have chosen calendar-based ranges as they are intuitive to data scientists and sysadmins as well as widely understood [11, 35, 51].

While different workloads will include a different mix of (age, length) queries, in our evaluation, we make no assumption about the relative predominance of query classes. In reality, for a given deployment, one typically expects to have some insight into the nature of the queries or their temporal distribution [49] which can be helpful.

Dataset. To simulate different real-world data streams, we generated synthetic data with three different arrival patterns: exponential Poisson with λ = 62.5 billion/year, and two heavy-tailed Pareto arrivals, with a finite mean and variance ($\alpha = 2.2$), and a finite mean but infinite variance ($\alpha = 1.2$); our parameter selection is influenced by well-known studies [22, 37]. Values are chosen uniformly at random from a finite range. Stream dataset totaled 1 PB on a single node with 1024 1TB streams each with 62.5 billion 16 byte entries (8-byte timestamp, 8-byte value).

For all microbenchmarks, we configure SummaryStore with four operators on each dimension: Count, Sum, Bloom filter, and Count-Min Sketch (CMS). Count and Sum are both 8-byte Longs; for the Bloom filter and the CMS, we chose a configuration with 5 hash functions, and a width of 1000; this corresponds to a 1% false-positive rate for the Bloom filter [26] and a SummaryStore window size of roughly 40KB. Note that selecting a larger Bloom filter width will lead to a corresponding increase in window size.

Results. Figure 9 presents a series of heatmaps for the described setup. The heatmaps show all three evaluation metrics for the four query operators; all heatmaps are from queries on the same instance of SummaryStore, providing a composite view of accuracy, performance, and storage compaction. Each heatmap is annotated with the query (*e.g.*, Count) on top left, the metric (*e.g.*, error) at top center, and the degree of compaction on top right. The x- and y-axes show query age and length respectively. Each axis is divided into 4 (age, length) classes, for a total of 16 cells; each cell shows a value and is shaded based on the value – the darker the shade, the higher the value. For all three metrics, namely, error, latency, and CI width, the larger the value (the darker the shade) the worse it is. Note that the errors are affected by the size of *individual* streams within a SummaryStore instance, which in this case is 1TB, while query latency is affected by the total size of the store *across* streams (1 PB).

Error: We first present the results for the infinite variance Pareto streams ($\alpha < 2$) which are extremely unpredictable and represent a pathological case for SummaryStore's summarization; shown in the first row of Figure 9.

Each cell represents the 95%-tile error for a repeated run of 1000 queries all belonging to the same (age, length) class and is thus statistically significant. For Count and Sum, most cells are white, implying errors less than 10%; in fact, most errors are below 5%. At smaller age, the proportional statistical response proves effective. The high data arrival rate leads to a large number of events packed into windows; this makes sub-window queries more predictable and considerably reduces errors. The queries that do perform poorly, shaded darker, are ones that pose an age in months, going back to severely decayed data, and a length in minutes or hours, a relatively tiny fraction of a potentially large-span window; this is as expected, particularly given the infinite variance arrivals which stretch the limits of the sub-window statistical techniques.

Most Bloom filter queries are accurate, except for month-age queries, similar to Count and Sum. The answer turns out to almost always be `true`: since we generate values uniformly from a finite set at a high arrival rate, over long time spans every value is very likely to be seen at least once. However, the *frequencies* of individual values exhibit significant variability [25] which affects the CMS, where we find errors of up to 5% except, again, the month-age queries.

CI width: As with errors, the CI width for Count, Sum and CMS is small for non-month-aged queries over small lengths. Bloom filter CIs are wide because at this aggressive level of compaction, SummaryStore needs to maintain very wide windows covering millions of elements each, and in the absence of additional window metadata it is not possible for the Bloom filter to localize the true results.

Latency: As can be seen, cold-cache latencies are < 1 minute in the 95th %-ile in all query ranges. Latencies are largest in queries scanning days and months (read >10% of a stream).

Other stream arrivals. Infinite variance Pareto streams are an extreme. Figure 10 shows the error and CI width for more typical Poisson arrivals; latency is quite comparable to Figure 9 and hence omitted. Uniformly low errors demonstrate the effectiveness of SummaryStore's statistical techniques, which in this case benefit from a more predictable arrival process. CI width significantly improved for all but Bloom filters; the reason again lies in the underlying time span of SummaryStore's windows at this massive scale. For finite variance Pareto streams $\alpha = 2.2$, observed results (not shown) were similar to Poisson with marginally higher errors and CI widths.

Takeaway. These results collectively demonstrate that SummaryStore can indeed scale to "colossal" streaming data; its statistical techniques are effective and accurate, for most queries, under high degrees of compaction; there do exist a few large-age small-length queries that perform poorly.

7.3 Confidence stimates at lower compaction

7.3.1 CI widths at 5x compaction. Figure 11 shows only CI width heatmaps for a lower-velocity Poisson stream; while we omit error and latency heatmaps for brevity, errors continue to stay low and latency does not exceed 100ms. In this experiment, the focus is on the source of the one remaining imprecision, the CI widths. We present results for a more moderate 5x compaction using PowerLaw(1,1,2,1). Note that this is for a smaller 5 GB/stream dataset generated by a lower-velocity Poisson process ($\lambda = 10/s$). The same experimental setup, and compaction, but with Exponential(2,142,1) decay yielded strictly worse errors ranging from 10% to 200% relative to PowerLaw (not shown).

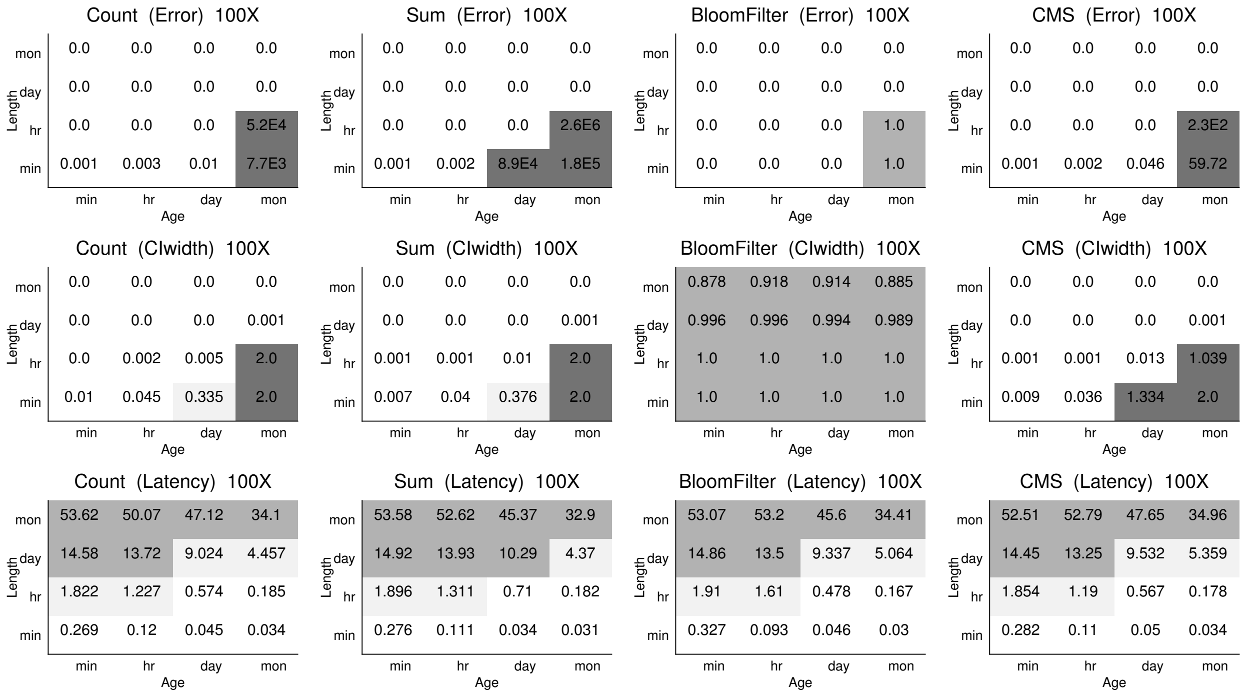


Figure 9: Query error, CI widths, latency for 1 PB (1024 x 1 TB) data streams; MicroBenchmarks: Count, Sum, Bloom filter, CMS. Decay PowerLaw(1,1,1,1) with ~100x compaction. Pareto arrivals ($\alpha = 1.2$) with ∞ variance. For all, larger values and darker shading are worse; for errors and CI width, unshaded cells imply value $< 10\%$. Latency in seconds. x-axis bins: Age, y-axis: Length.

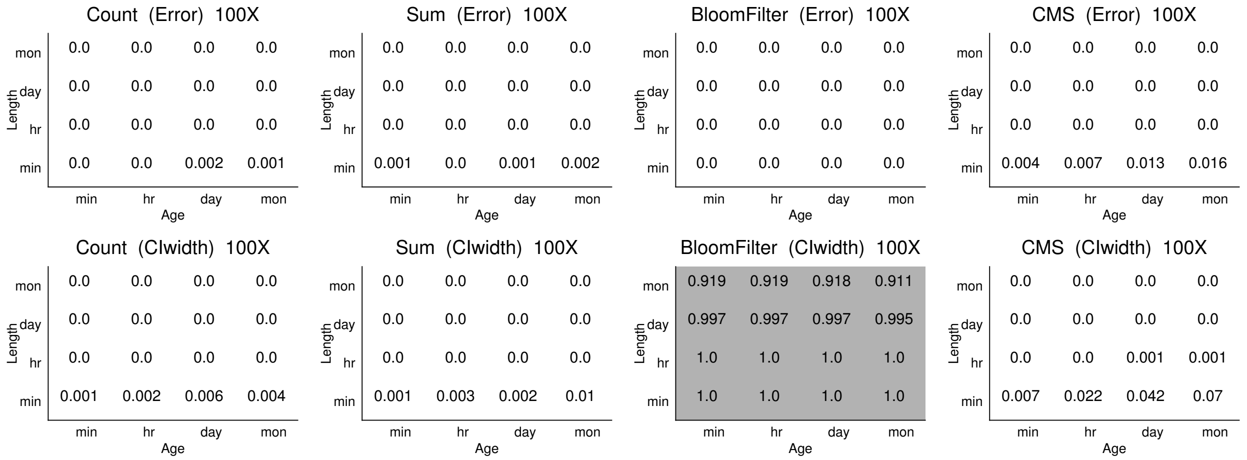


Figure 10: Query error, CI widths for 1 PB (1024 x 1 TB) data streams. Change with Fig 9 is Poisson arrivals; similar latency.

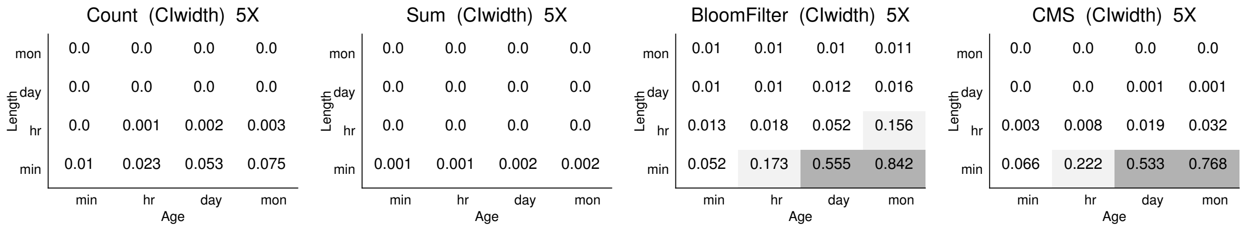


Figure 11: CI widths for MicroBenchmarks. Slow Velocity, Lower Compaction (5x) w/ Decay PowerLaw(1,1,2,1); Poisson ($\lambda = 10$).

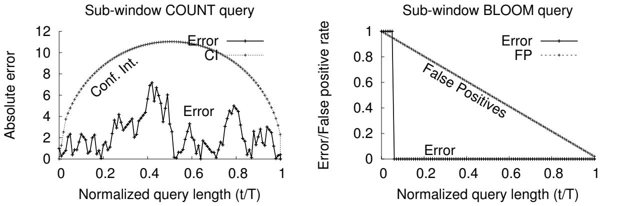
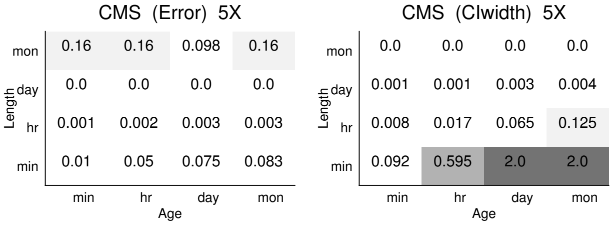


Figure 12: Sub-window answers and error estimates. Error and CI widths in Count and Bloom filter as length of sub-window query varies from zero to full window length



CMS (Error) 5X

Length \ Age	min	hr	day	mon
mon	0.16	0.16	0.098	0.16
day	0.0	0.0	0.0	0.0
hr	0.001	0.002	0.003	0.003
min	0.01	0.05	0.075	0.083

CMS (CIwidth) 5X

Length \ Age	min	hr	day	mon
mon	0.0	0.0	0.0	0.0
day	0.001	0.001	0.003	0.004
hr	0.008	0.017	0.065	0.125
min	0.092	0.595	2.0	2.0

Figure 13: Query error and CI widths for MLab. Decay PowerLaw(1,1,4,1) giving 5X compaction; we omit latency for brevity.

With the reduced decay, SummaryStore generates a higher number of windows, reducing their average time-spans. Since the CI upper bound tracks the largest window spans, its reduction improves the confidence estimates, especially for Bloom filter, despite the higher stochastic variability introduced by the sparser (lower-velocity) arrivals.

CI widths for all Count and Sum queries are sufficiently low, the maximum being 7.5%. All but two cells for Bloom filter have widths less than 17%, the two exceptions being 56% and 84%; a marked improvement from Figures 9, 10.

7.3.2 Understanding the profile of error estimates.

To further explain the estimation procedure in sub-window queries, Figure 12 shows how error varies with increasing query lengths for Count and Bloom filter; specifically, we keep query age constant at 0, aligned with a window boundary, and vary length t from 0 to the full window length T.

In the Count query, note that both empirical error and CI width are largest near the middle of the interval and fall to 0 near either edge. This is because estimating the count for the sub-window $[T_1, T]$ given the count for $[T_1, T_2]$ requires the same information as estimating the count for $[T, T_2]$: what matters is the (minimum) distance to either end of the interval, which is largest for queries that overlap exactly half the interval. CI widths show an elliptical profile proportional to $\sqrt{t/T * (1 - t/T)}$ (§5).

In Bloom filters, a similar symmetry does not result since, by definition, Bloom filter queries are concerned solely with the existence of a given value in the time-interval they specify; instead, error depends on the absolute amount of overlap the interval has with the window. False Positive probability gradually falls as overlap increases, asymptoting out to the inherent False Positive rate the Bloom filter is configured with (1% in this case) for a full-window query.

7.4 Application traces

M-Lab network traffic traces. We repeat the workload Cui *et al.* used to evaluate Aperture [38], running frequency queries (using a Count-Min sketch) to estimate visit frequencies of various IP address in an M-Lab [38, 44] log.

Dataset. The entire Paris traceroute data collected by M-Lab [44] in 2015, a log of 170M visits over a 1 year period.
Results. workload consists of frequency queries (using a CMS) on various IP addresses in a log of client visits. Figure 13 shows error and CI width results at 5X compaction, binned by query age and length. Error is less than 16% in all minute-or-longer query classes, and less than 10% in all but three. Second-length queries see errors up to 44%. CI width stays below 13% in all but three minute-or-longer classes. Latencies (not shown) were low, <30 ms for 95th %-ile.

IBM Tivoli Storage Manager backup traces. TSM retains logs of actions performed in the system. We evaluate a workload of queries on this log collated by an IBM sysadmin [11], consisting of a combination of sum, count and frequency queries on time intervals of varying ages and lengths between 1 day and 1 month (*e.g.*, how many bytes did node 7 upload over the past week?).
Dataset. We generated a corresponding synthetic dataset simulating a population of 10,000 nodes performing storage backups once every hour over a period of 7 years, each backup failing with a certain probability (we set failure rate to 1%), with backup sizes as per Wallace *et al.*[82].
Results. On all TSM queries we tested, we observed low errors, with tight CIs, since the queries were all at fairly coarse granularities (length at least a day). At 5X compaction, error in all query classes was less than 2%, with the worst error arising on queries with age = years, length = days. We omit detailed per-query results for brevity.

8 Related Work

8.1 Time-Series Storage Systems

Most time–series stores [7, 18, 46, 50, 66, 74] enumerate raw data. As such, the size of the stores grows linearly (or worse) with data ingest and query latency is adversely affected. Some of these stores [18, 66] maintain additional aggregations and roll-ups to reduce query latency, further increasing the cost to store data. These stores thus can write and read large volumes of data but at significant storage cost.

8.2 Approximate-Query-Processing (AQP) Systems

AQP systems are the closest to SummaryStore in the set of tradeoffs – performance and freshness [30], response time and accuracy [17], and most notably, storage space and accuracy [38] – they offer for queries. AQP systems are becoming increasingly popular with the tremendous growth in data volumes and the need for low latency.

BlinkDB [17] approximates by sampling; it constructs multi-dimensional *stratified* samples, over-representing rare subgroups relative to the uniform random sample. BlinkDB shares SummaryStore's goal in providing bounded errors and low latency on large volumes of data but is not designed for time-series data. Follow-up work tackles the challenges of error estimation under sampling [16] and makes a strong case for why reliable error bars are critical to AQP systems; SummaryStore takes note of this challenge and proposes techniques to reliably estimate error under temporal decay.

Succinct [15] supports a limited set of search queries on strings stored in flat files; it compresses files using Compressed Suffix Arrays, and subsequent sampling on the arrays to achieve significant compression for specific workloads. BlowFish [55] extends Succinct but creates multiple levels of sampled arrays, for the same data, offering configurable tradeoff between storage size and performance. Both Succinct and BlowFish are suitable primarily for search queries on unstructured files which, while useful, offers narrow functionality. MacroBase [21] uses *damped* reservoir sampling [81] over exponentially-damped windows, with an emphasis on outlier detection in data streams; inline with this target use case, the windows maintain a heavy-hitter counting sketch data structure. QuickR [54] and JetStream [69] are also recently proposed AQP systems for big data and wide area, respectively, that leverage sampling and aggregation.

Broadly speaking, sampling is highly effective and has been widely adopted in AQP systems with success. SummaryStore demonstrates the utility of statistical summaries, in addition to samples, to provide bounded-error, bounded-capacity AQP on high-volume streaming data.

Similar to SummaryStore, Aperture [38] maintains aggregate statistics per window for low-latency time-series analysis but focuses on correlation search. Aperture treats all windows homogeneously (*i.e.*, no decay) and provides no temporal bias. It supports only window-aligned queries which is both limiting and side-steps the challenges in reliable error estimation. In spite of its limitations, Aperture is a step forward towards stores designed for temporal analytics.

8.3 Time-Decayed Expiration in Storage Systems

Position papers on a delete-optimized store [24, 43] were motivated by short object lifetimes in a streaming environment; the underlying assumption was that a substantial amount of data is written to disk, read few times, and then quickly deleted. For these ideas, the decay applied to the retention value and not to the data; files would be deleted in their entirety when the value eroded. Contemporary time–series stores support similar time expiration [8].

Palimpsest [73] is a store offering bounded duration of files: all storage is "soft capacity", with no *a priori* guarantees on the duration of persistence; files can be reclaimed automatically but are available and secure meanwhile. Unlike "best-effort" expiry, SummaryStore decays the physical allocation on a continuous basis through a predictable, algorithmic process to control query errors. Hyperion [42] shares the motivation of supporting retrospective querying on archived data streams but emphasizes on performance.

8.4 Algorithmic Time-Decayed Representation

The Exponential Histogram (EH) [39] maintains exponentially growing spans of time $(1, 2, 4, \cdots, 2^{\lceil \log N \rceil})$ for a sliding window of N time units; data within a range is aggregated as counts. While EH can be generalized to counting beyond the sliding window, its construction forces an unacceptably-aggressive decayed representation of data. The Weight-Based Merging Histogram (WBMH) [32] generalizes EH to track arbitrary time-weighted counts in integer streams. They show that for many weight functions it suffices to maintain an even more aggressive windowing than in EH; in particular, they show how to track approximate polynomially weighted counts using power-of-k windowing with $k \geq 2$ – the basis for SummaryStore's merge process.

9 Conclusions

While large volumes of time-series data are driving powerful analyses, storage systems are lagging behind. We have built SummaryStore, the first of its kind approximate time–series store that provides unprecedented (100x) cost savings for data storage; we developed and implemented novel techniques to ingest and decay data, and provide accurate query responses on significantly summarized data. SummaryStore compacted a 1 PB dataset to 10 TB while answering 95% of queries with less than 5% error, with median and worst-case cold-cache latency 1.3s and 70s respectively. It was also able to run real-world applications for forecasting and anomaly detection on 6-10x compacted data while preserving the fidelity of the analyses.

Acknowledgments

We thank Doug Terry and Masoud Saeida Ardekani for excellent feedback throughout, Moises Goldszmidt and Remzi Arpaci-Dusseau for helpful comments, our SOSP reviewers for thorough reviewing, and Aditya Akella for shepherding.

A Decay function properties (described in Table 4)

We now establish how the number of windows W used to represent a stream grows as a function of the number of elements N appended to the stream, for the two decay function families in Table 4. The other properties in Table 4 follow immediately: the size of the store (column 3) is simply W × the size of each window (which is a constant for a given stream in SummaryStore); and the marginal storage alloted to the nth oldest element (column 4) is the derivative of W with respect to N.

Power law decay. Consider the stream at the point when, for some K, the windows have lengths

$$\text{for } k = 1, 2, \ldots, K : R\ k^{p-1} \text{ of length } S\ k^q$$

and all windows are full. At this point, the total number of windows is

$$\begin{aligned} W &= R\left(1^{p-1} + 2^{p-1} + \ldots + K^{p-1}\right) \\ &= R\left(\frac{K^p}{p} + o\left(K^p\right)\right) \end{aligned}$$

and the total number of elements in the stream is

$$\begin{aligned} N &= RS\left(1^{p+q-1} + 2^{p+q-1} + \ldots + K^{p+q-1}\right) \\ &= RS\left(\frac{K^{p+q}}{p+q} + o\left(K^{p+q}\right)\right) \end{aligned}$$

Approximating by assuming a large stream and ignoring the lower-order terms, and eliminating the variable K, we get

$$W \approx \frac{(p+q)^{\frac{p}{p+q}}}{p} R\left(\frac{N}{RS}\right)^{\frac{p}{p+q}}$$

Exponential decay. Proceeding as with power law decay, consider the state of the stream at the point when the windows have lengths

$$\text{for } k = 1, 2, \ldots, K : R \text{ of length } Sb^k$$

and all windows are full. At this point, the number of windows is given by

$$W = RK$$

and the total number of elements in the stream is

$$\begin{aligned} N &= RS\left(b + b^2 + \ldots + b^K\right) \\ &= RS\frac{b^{K+1} - 1}{b - 1} \end{aligned}$$

Eliminating the variable K, we get

$$W = R\left(\log_b\left((b-1)\frac{N}{RS} + 1\right) - 1\right)$$

B Proofs for error estimators (described in §5)

To simplify notation, we present a sub-window query on a single window (Figure 14) in this section; the results extend naturally to queries spanning multiple windows.

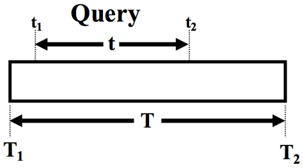


Figure 14: Sub-window query.

THEOREM B.1. *In the absence of additional information localizing arrivals to specific positions in the window, given that count*$[T_1, T_2] = C$*, the expected value of count*$[t_1, t_2]$ *is* $C * t/T$.

PROOF. Divide $[T_1, T_2]$ into T/ϵ equally spaced sub-intervals of length ϵ each. In the absence of localizing information, the expected count of arrivals in each sub-interval must be identical. Given that there were C arrivals in total in $[T_1, T_2]$, by linearity of expectation the expected count of arrivals in each sub-interval is $C/(T/\epsilon) = \epsilon C/T$. Since $[t_1, t_2]$ overlaps between $\lfloor t/\epsilon \rfloor$ and $\lceil t/\epsilon \rceil$ of these sub-intervals, the expected value of count$[t_1, t_2]$ approaches $(t/\epsilon) * (\epsilon C/T) = C * t/T$ as $\epsilon \to 0$. □

THEOREM B.2. *In a Poisson stream*

$$\Pr(count[t_1, t_2] \mid count[T_1, T_2] = C) \sim Binom(C, \mathtt{t/T})$$

PROOF. In a Poisson stream, conditioned on the fact that there were C arrivals in $[T_1, T_2]$, the positions of the arrivals have the same statistical distribution [36] as picking C independent uniform samples from $[T_1, T_2]$, each of which has a t/T chance of being inside the sub-interval $[t_1, t_2]$. Therefore the distribution count$[t_1, t_2] \sim$ Binom$(C, t/T)$ with mean $C * t/T$ and variance $C * t/T * (1 - t/T)$. □

THEOREM B.3. *In a stream generated by a renewal-reward process [36] with* `i.i.d.` *interarrivals with mean* μ_t *and standard deviation* σ_t*, and* `i.i.d.` *values with mean* μ_v *and standard deviation* μ_v*, given that count*$[T_1, T_2] = C$ *and sum*$[T_1, T_2] = S$*, in the limit as window sizes → ∞ the posterior distribution on sub-window count and sum approach the bivariate normal distribution*

$$N\left(\frac{t}{T}\begin{bmatrix} C \\ S \end{bmatrix}, \frac{t}{T}\left(1 - \frac{t}{T}\right)\frac{T}{\mu_t^3}\begin{bmatrix} \sigma_t^2 & \sigma_t^2\mu_v \\ \sigma_t^2\mu_v & (\sigma_t^2\mu_v^2 + \sigma_v^2\mu_t^2) \end{bmatrix}\right)$$

PROOF. Define the random variables

$$\begin{aligned} (C_t, S_t) &:= (\text{count}[t_1, t_2], \text{sum}[t_1, t_2]) \\ (C_T, S_T) &:= (\text{count}[T_1, T_2], \text{sum}[T_1, T_2]) \end{aligned}$$

Standard renewal-theoretic approximations [36] state that in the limit as window sizes go to infinity

- (C_T, S_T) is bivariate normal with

$$\mathbb{E}[C_T] = \frac{T}{\mu_t} \qquad \mathbb{E}[S_T] = \frac{\mu_v T}{\mu_t}$$

$$\mathrm{Var}[C_T] = \frac{\sigma_t^2 t}{\mu_t^3} \qquad \mathrm{Var}[S_T] = \frac{(\sigma_t^2 \mu_v^2 + \sigma_v^2 \mu_t^2)T}{\mu_t^3}$$

$$\mathrm{Cov}[C_T, S_T] = \frac{\sigma_t^2 \mu_v t}{\mu_t^3}$$

- The sum and count in $[t_1, t_2]$ are independent of the sum and count in $[T_1, t_1]$ and $[t_2, T_2]$. (They are weakly coupled by the lone arrivals going over the interval boundaries t_1 and t_2, whose effect becomes negligible in the large window limit.)

Applying these two limiting approximations, and simplifying, it can be shown that the joint distribution of the random variables (C_t, S_t, C_T, S_T) is multivariate normal with mean

$$\bar{\mu} = \begin{bmatrix} \frac{t}{\mu_t} \\ \frac{\mu_v t}{\mu_t} \\ \frac{T}{\mu_t} \\ \frac{\mu_v T}{\mu_t} \end{bmatrix}$$

and covariance matrix

$$\bar{\Sigma} = \frac{1}{\mu_t^3} \begin{bmatrix} \sigma_t^2 t & \sigma_t^2 \mu_v t & \sigma_t^2 t & \sigma_t^2 \mu_v t \\ \sigma_t^2 \mu_v t & (\sigma_t^2 \mu_v^2 + \sigma_v^2 \mu_t^2)t & \sigma_t^2 \mu_v t & (\sigma_t^2 \mu_v^2 + \sigma_v^2 \mu_t^2)t \\ \sigma_t^2 t & \sigma_t^2 \mu_v t & \sigma_t^2 T & \sigma_t^2 \mu_v T \\ \sigma_t^2 \mu_v t & (\sigma_t^2 \mu_v^2 + \sigma_v^2 \mu_t^2)t & \sigma_t^2 \mu_v T & (\sigma_t^2 \mu_v^2 + \sigma_v^2 \mu_t^2)T \end{bmatrix}$$

Defining

$$K := \frac{1}{\mu_t^3} \begin{bmatrix} \sigma_t^2 & \sigma_t^2 \mu_v \\ \sigma_t^2 \mu_v & (\sigma_t^2 \mu_v^2 + \sigma_v^2 \mu_t^2) \end{bmatrix}$$

this simplifies to

$$\bar{\Sigma} = \begin{bmatrix} tK & tK \\ tK & TK \end{bmatrix}$$

Now conditioning on the knowledge that $C_T = C$ and $S_T = S$, $(C_t, S_t \mid C_T = C, S_T = S)$ is again bivariate normal [12] with mean

$$\begin{bmatrix} \frac{t}{\mu_t} \\ \frac{\mu_v t}{\mu_t} \end{bmatrix} + (tK)(TK)^{-1} \left(\begin{bmatrix} C \\ S \end{bmatrix} - \begin{bmatrix} \frac{T}{\mu_t} \\ \frac{\mu_v T}{\mu_t} \end{bmatrix} \right) = \begin{bmatrix} \frac{t}{T} C \\ \frac{t}{T} S \end{bmatrix}$$

and covariance

$$tK - (tK)(TK)^{-1}(tK) = \frac{t}{T}\left(1 - \frac{t}{T}\right) TK \qquad \square$$

The proof for count can be adapted to non-numeric streams: we would simply consider a renewal process (without associated reward) and restrict to considering the joint distribution of (C_t, C_T).

Theorem B.3 subsumes Theorem B.2 in the large window limit: in a Poisson stream, $\sigma_t / \mu_t = 1$, and a standard result [57] states that as the number of arrivals goes to infinity the binomial distribution converges to the normal distribution with the same mean and variance.

Theorem B.4. *In a Poisson stream of* `i.i.d.` *values, suppose a frequency summary suggests V occurrences of a value v in $[T_1, T_2]$. Then* $\Pr(v \in [t_1, t_2]) = 1 - (1 - t/T)^V$.

Proof. Theorem B.2 showed that $\Pr(count[T_1, T_2]) \sim \mathrm{Binom}(N, \mathtt{t/T})$. Now

$$\begin{aligned} \Pr(V \in [t_1, t_2]) &= \Pr(\text{at least one occurence in } [t_1, t_2]) \\ &= 1 - \Pr(\text{no occurences in } [t_1, t_2]) \\ &= 1 - \Pr(\mathrm{Binom}(V, \mathtt{t/T}) = 0) \\ &= 1 - (1 - \mathtt{t/T})^V \end{aligned} \qquad \square$$

Theorem B.5. *In a stream generated by a renewal-reward process with renewal-independent rewards, suppose $count[T_1, T_2] = C$ and $freq(v)[T_1, T_2] = V$. In the limit as window sizes $\to \infty$, the posterior distribution on sub-window frequency approaches*

$$freq(v)[t_1, t_2]) \to Hypergeom(C, V, C_t)$$

where $C_t \sim$ the posterior distribution over $count[t_1, t_2]$ from Theorem B.3.

Proof. There were C arrivals in $[T_1, T_2]$, C_t of which were in $[t_1, t_2]$, and V of which had value v. Since the stream is `i.i.d.`, the distribution over the count of arrivals which were both in $[t_1, t_2]$ and of value v has a hypergeometric shape [57]. □

Note that V might itself be a random variable, if a statistical characterization of the underlying frequency summary is available. The expression $\mathrm{Hypergeom}(C, V, C_t)$ denotes a compound distribution.

Corollary B.6. *Under the same assumptions as Theorem B.5, in the limit as window sizes $\to \infty$, the answer to the sub-window existence query $exists(v)[t_1, t_2]$ approaches*

$$Pr(v \in [t_1, t_2]) \to Pr\left(Hypergeom(C, V, C_t) > 0\right)$$

Proof. Follows from Theorem B.5 since

$$v \in [t_1, t_2] \iff \mathrm{freq}(v)[t_1, t_2] > 0 \qquad \square$$

References

[1] Green Button Initiative. http://www.greenbuttondata.org.

[2] Protocol Buffers. http://code.google.com/p/protobuf.

[3] Stratified sampling. Wikipedia. Posted at https://en.wikipedia.org/wiki/Stratified_sampling#Disadvantages.

[4] Three sigma rule. Wikipedia. Posted at https://en.wikipedia.org/wiki/Three_sigma_rule.

[5] Introducing Kale. https://codeascraft.com/2013/06/11/introducing-kale/, 2013.

[6] Ganglia Monitoring System. http://ganglia.info/, 2016.

[7] IBM Informix TimeSeries. https://www-01.ibm.com/software/data/informix/timeseries, 2016.

[8] InfluxDB: Downsampling and Data Retention. https://docs.influxdata.com/influxdb/v0.9/guides/downsampling_and_retention/, 2016.

[9] Netflix Atlas time-series telemetry platform. https://github.com/Netflix/atlas, 2016.

[10] Netflix Edda. https://github.com/Netflix/edda, 2016.

[11] SQL for IBM Tivoli Storage Manager. http://thobias.org/tsm/sql/, 2016.

[12] Multivariate normal distribution. https://en.wikipedia.org/wiki/Multivariate_normal_distribution, 2017.

[13] G. Adomavicius, R. Sankaranarayanan, S. Sen, and A. Tuzhilin. Incorporating contextual information in recommender systems using a multidimensional approach. *ACM Transactions on Information Systems (TOIS)*, 23(1):103–145, 2005.

[14] P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, and K. Yi. Mergeable summaries. *ACM Transactions on Database Systems (TODS)*, 38(4):26, 2013.

[15] R. Agarwal, A. Khandelwal, and I. Stoica. Succinct: Enabling queries on compressed data. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 337–350, 2015.

[16] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica. Knowing when you're wrong: building fast and reliable approximate query processing systems. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 481–492. ACM, 2014.

[17] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42. ACM, 2013.

[18] M. P. Andersen and D. E. Culler. Btrdb: optimizing storage system design for timeseries processing. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 39–52, 2016.

[19] C. M. Antunes and A. L. Oliveira. Temporal data mining: An overview. In *KDD workshop on temporal data mining*, volume 1, page 13, 2001.

[20] Apache. Apache Commons Math library. http://commons.apache.org/proper/commons-math/, 2016.

[21] P. Bailis, D. Narayanan, and S. Madden. Macrobase: Analytic monitoring for the internet of things. *arXiv preprint arXiv:1603.00567*, 2016.

[22] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *ACM SIGMETRICS Performance Evaluation Review*, volume 26, pages 151–160. ACM, 1998.

[23] R. M. Bell and Y. Koren. Lessons from the netflix prize challenge. *ACM SIGKDD Explorations Newsletter*, 9(2):75–79, 2007.

[24] R. Bhagwan, F. Douglis, K. Hildrum, J. O. Kephart, and W. E. Walsh. Time-varying management of data storage. In *Workshop on Hot Topics in System Dependability (HotDep)*, 2005.

[25] L. Bhuvanagiri and S. Ganguly. *Estimating Entropy over Data Streams*, pages 148–159. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[26] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[27] A. Bremler-Barr, E. Cohen, H. Kaplan, and Y. Mansour. Predicting and bypassing end-to-end internet service degradations. *IEEE Journal on Selected Areas in Communications*, 21(6):961–978, 2003.

[28] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995.

[29] P. G. Campos, F. Díez, and I. Cantador. Time-aware recommender systems: a comprehensive survey and analysis of existing evaluation protocols. *User Modeling and User-Adapted Interaction*, 24(1-2):67–119, 2014.

[30] J. Cipar, G. Ganger, K. Keeton, C. B. Morrey III, C. A. Soules, and A. Veitch. Lazybase: trading freshness for performance in a scalable database. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 169–182. ACM, 2012.

[31] E. Cohen, H. Kaplan, and J. Oldham. Managing tcp connections under persistent http. *Computer Networks*, 31(11):1709–1723, 1999.

[32] E. Cohen and M. Strauss. Maintaining time-decaying stream aggregates. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 223–233. ACM, 2003.

[33] G. Cormode, F. Korn, and S. Tirthapura. Time-decaying aggregates in out-of-order streams. In *Proceedings of the Twenty-seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '08, pages 89–98, New York, NY, USA, 2008. ACM.

[34] C. Cortes and D. Pregibon. Giga-mining.

[35] Cory Watson. Observability at Twitter. https://blog.twitter.com/2013/observability-at-twitter, 2013.

[36] D. R. Cox. *Renewal theory*. Methuen, 1962.

[37] M. E. Crovella and A. Bestavros. Self-similarity in world wide web traffic: evidence and possible causes. *IEEE/ACM Transactions on networking*, 5(6):835–846, 1997.

[38] H. Cui, K. Keeton, I. Roy, K. Viswanathan, and G. R. Ganger. Using data transformations for low-latency time series analysis. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 395–407. ACM, 2015.

[39] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM journal on computing*, 31(6):1794–1813, 2002.

[40] S. Dawson-Haggerty, A. Krioukov, J. Taneja, S. Karandikar, G. Fierro, N. Kitaev, and D. E. Culler. Boss: Building operating system services. In *NSDI*, 2013.

[41] Dennis Shasha. Time Series in Finance: the array database approach. http://cs.nyu.edu/shasha/papers/jagtalk.html, 1997.

[42] P. Desnoyers and P. J. Shenoy. Hyperion: High volume stream archival for retrospective querying. In *USENIX Annual Technical Conference*, pages 45–58, 2007.

[43] F. Douglis, J. Palmer, E. S. Richards, D. Tao, W. H. Tetzlaff, J. M. Tracey, and J. Yin. "position: short object lifetimes require a delete-optimized storage system". In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 6. ACM, 2004.

[44] C. Dovrolis, K. Gummadi, A. Kuzmanovic, and S. D. Meinrath. Measurement lab: overview and an invitation to the research community. *ACM SIGCOMM Computer Communication Review*, 40(3):53–56, 2010.

[45] Federal Reserve Economic Data. https://en.wikipedia.org/wiki/Federal_Reserve_Economic_Data, 2017.

[46] T. Gupta, R. P. Singh, A. Phanishayee, J. Jung, and R. Mahajan. Bolt: Data management for connected homes. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 243–256, 2014.

[47] E. Hailemariam, R. Goldstein, R. Attar, and A. Khan. Real-time Occupancy Detection Using Decision Trees with Multiple Sensor Types. In *Proc. of Symposium on Simulation for Architecture and Urban Design*, 2011.

[48] Price Trends: Internal Hard Drives. https://pcpartpicker.com/trends/internal-hard-drive/.

[49] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in olap data cubes. In *In Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, 1997.

[50] InfluxData. Influxdb time-series database. http://influxdb.com, 2015.

[51] K. J. Jacob and D. Shasha. Fintime – a financial benchmark.

[52] Jamie Wilkinson. Google Prometheus: A practical guide to alerting at scale. https://docs.google.com/presentation/d/1X1rKozAUuF2MVc1YXElFWq9wkcWv3Axdldl8LOH9Vik/edit#slide=id.g598ef96a6_0_341, 2016.

[53] Jason Kincaid. Facebook EdgeRank: The Secret Sauce That Makes Facebook's News Feed Tick. https://techcrunch.com/2010/04/22/facebook-edgerank, 2010.

[54] S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. 2016.

[55] A. Khandelwal, R. Agarwal, and I. Stoica. Blowfish: dynamic storage-performance tradeoff in data stores. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 485–500, 2016.

[56] Y. Koren. Collaborative filtering with temporal dynamics. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 447–456. ACM, 2009.

[57] D. M. Lane. Online statistics education: A multimedia course of study. http://onlinestatbook.com/.

[58] F. Lautenschlager, M. Philippsen, A. Kumlehn, and J. Adersberger. Chronix: Long term storage and retrieval technology for anomaly detection in operational data. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 229–242, Santa Clara, CA, 2017. USENIX Association.

[59] D. Leary. Nimble Storage Blog: Why Does Enterprise Storage Cost So Much? . https://www.nimblestorage.com/blog/why-does-enterprise-storage-cost-so-much/, 2010.

[60] L. Li, W. Chu, J. Langford, and R. E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*, pages 661–670. ACM, 2010.

[61] Mike Keane. 1.5 million Log Lines per Second. http://www.bigdataeverywhere.com/files/chicago/BDE-15millionLogLinesPerSecond-KEANE.pdf, 2014.

[62] T. Mitchell. Never-ending learning. Technical report, DTIC Document, 2010.

[63] M. Natrella. NIST/SEMATECH e-handbook of statistical methods. 2010.

[64] NOAA Global Surface Summary of the Day Weather Data. https://cloud.google.com/bigquery/public-data/noaa-gsod.

[65] R. Peck, C. Olsen, and J. L. Devore. *Introduction to statistics and data analysis*. Cengage Learning, 2015.

[66] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan. Gorilla: a fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, 8(12):1816–1827, 2015.

[67] Power Law. https://en.wikipedia.org/wiki/Power_law.

[68] Prophet: forecasting at scale. https://research.fb.com/prophet-forecasting-at-scale/, Feb 2017.

[69] 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 *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 275–288, 2014.

[70] S. Reddy, M. Mun, J. Burke, D. Estrin, M. Hansen, and M. Srivastava. Using Mobile Phones To Determine Transportation Modes. *ACM Transactions on Sensor Networks (TOSN)*, 2010.

[71] C. Reiss, J. Wilkes, and J. L. Hellerstein. Google cluster-usage traces: format + schema. Technical report, Google Inc., Mountain View, CA, USA, Nov. 2011. Revised 2014-11-17 for version 2.1. Posted at https://github.com/google/cluster-data.

[72] RocksDB. RocksDB. http://rocksdb.org/, 2016.

[73] T. Roscoe and S. Hand. Palimpsest: Soft-capacity storage for planetary-scale services. In *HotOS*, pages 127–132, 2003.

[74] B. Sigoure. Opentsdb scalable time series database (tsdb). http://opentsdb.net, 2012.

[75] R. P. Singh, S. Keshav, and T. Brecht. A Cloud-based Consumer-centric Architecture for Energy Data Analytics. In *Proc. of ACM e-Energy, 2013*.

[76] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[77] S. J. Taylor and B. Letham. Facebook open source project: Forecasting at scale. https://github.com/facebookincubator/prophet, 2017.

[78] Ted Friedman. Gartner Report: "Internet of Things: Biggest Impact Ever on Information and Master Data" . http://www.gartner.com/webinar/3291728?srcId=1-7389946120, 2016.

[79] S. Thrun and L. Pratt. *Learning to learn*. Springer Science & Business Media, 2012.

[80] M. van Rijmenam. Self-driving cars will create 2 petabytes of data, what are the big data opportunities for the car industry? https://datafloq.com/read/self-driving-cars-create-2-petabytes-data-annually/172, 2017.

[81] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[82] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of backup workloads in production systems. In *FAST*, 2012.

[83] W. W.-S. Wei. *Time series analysis*. Addison-Wesley publ Reading, 1994.

[84] Wikipedia Traffic Statistics V2. https://aws.amazon.com/datasets/wikipedia-traffic-statistics-v2/.

[85] J. Wilkes. More Google cluster data. Google research blog. Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.

[86] Z. Yang, N. Li, B. Becerik-Gerber, and M. Orosz. A multi-sensor based occupancy estimation model for supporting demand driven hvac operations. In *Proceedings of the 2012 Symposium on Simulation for Architecture and Urban Design*, page 2. Society for Computer Simulation International, 2012.