Pebbles: Leveraging Sketches for Processing Voluminous, High Velocity Data Streams

Thilina Buddhika, Sangmi Lee Pallickara, and Shrideep Pallickara, *Members, IEEE*

Abstract—Voluminous, time-series data streams originating in continuous sensing environments pose data ingestion and processing challenges. We present a holistic methodology centered around data sketching to address both challenges. We introduce an order-preserving sketching algorithm that we have designed for space-efficient representation of multi-feature streams with native support for stream processing related operations. Observational streams are preprocessed at the *edges* of the network generating sketched streams to reduce data transfer costs and energy consumption. Ingested sketched streams are then processed using sketch-aware extensions to existing stream processing APIs delivering improved performance. Our benchmarks with real-world datasets show up to a $\sim$8$\times$ reduction in data volumes transferred and a $\sim$27$\times$ improvement in throughput.

Index Terms—data sketches, stream processing systems, edge computing, internet-of-things

1 Introduction

By the year 2020, 30 billion networked devices are expected to be in operation, serving a population of about 7.6 billion [1]. Most of these devices are equipped with one or more sensing capabilities. A confluence of factors has contributed to dramatic increases in the number of connected sensing devices. These contributing factors include inter alia advances in miniaturization, falling costs, networking enhancements, and sustained improvements in the quality and capacity of battery technologies. These sensing devices are now deployed in *continuous sensing environments (CSEs)* where phenomena of interest are monitored at ever increasing precisions and frequencies. CSEs arise in settings such as Internet-of-Things (IoT), Cyber Physical Systems, and mission critical monitoring systems. Domains that CSEs have been deployed include atmospheric and ecological monitoring [2], traffic [3], environmental monitoring [4], health care [5], [6], and industrial equipment monitoring [7] among others. Majority of the data streams originating in such settings contain a logical or an absolute temporal component [8].

Sensors and actuators form the lowest layer in a typical CSE architecture [9]. Edge devices may have on-board sensors whose measurements they report, which is the case with mobile phones and smart wearables such as active trackers. Alternatively, the edge devices may act as *data mules* [10] where the data is pulled from the sensors periodically using limited-range, custom transport protocols such as PSFQ [11] and ESRT [12] used in sensor networks. The edge device consolidates readings from multiple sensors into a *single timestamped observation* capturing multiple aspects of a monitored entity at a given point in time. Constructed streams can be processed at a central location, at the edges of the network in proximity to stream sources (edge processing), or using a hybrid of the previous two approaches (federated processing). With centralized processing, the data is transferred to a central location, also known as the *data sink* or the *middleware layer*, for processing [10]. In some IoT reference architectures, the observational streams are transferred to private/public clouds through cloud gateways [9], [13]. The transferred data can be either processed in near-realtime as streams, or stored and subsequently processed using batch processing systems. Edge and federated processing leverages the limited capabilities at the edges of the network to perform less resource intensive processing. In this study, we focus on near-realtime processing of data streams.

1.1 Challenges

The challenges involved with these architectures are two fold: (1) data stream ingestion, and (2) data stream processing. These challenges can be attributed to the vast volumes of data produced in CSE settings by a large number of devices alongside the ability of each device to produce data at high rates. Data stream ingestion is challenging due to:

- *Power constraints* - Most edge devices are battery powered or have limited power profiles. Communication is the dominant energy consuming task on these devices [14], [10], [15]; transferring voluminous data can become infeasible.
- *Limited bandwidth* - Edge devices are connected to the remainder of the data ingestion pipeline via wide area networks with limited bandwidth [16].
- *Data transfer costs* - When the processing middleware is operating in public clouds, users are billed for the volume of data transferred across data center network boundaries. This is in addition to the bandwidth costs for the data transfer.

Challenges involved with data processing include:

- *Processing costs* - The amount of resources required for executing stream processing middleware is proportional to the volume of the data. Users are thus looking at ever increasing processing demands.
- *Reprocessing past data* - Changes in business requirements, discovery of bugs entail reprocessing *past* data using the updated versions of the applications [17], [18]. This entails storing data streams for extended periods of time.

With the ability to add limited processing and storage capabilities to the edge devices, and decentralized data processing initiatives such as cloudlets [19], distributed telco clouds [20], and fog computing [21], several attempts have been made to preprocess/process data at the edges and reduce data transfer to the center [16], [22], [14], [23], [24], [25], [26], [27], [28], [29], [30], [10], [31]. There is a rich body of work on improving stream processing at the center through efficient resource management and scheduling of operators. These solutions only partially address the data ingestion and processing challenges and have the following drawbacks.

- *Limited applicability* - The applicability of federated stream processing is constrained due to the limited processing power

and storage capabilities of edge devices, and non-local data dependencies.
- *Poor support for multi-feature streams* - Majority of the data reduction techniques are designed for single-feature streams. Applicability to multi-feature streams entails expensive stream joins at the center.
- *Focuses only on reducing the data transfer* - Most data reduction techniques focus only reducing the data transfer and reconstruct the entire stream at the center [32], and do not address the data processing challenges.
- *Considering input streams as transient* - Existing edge processing technique are driven by the current user requirements and consider data streams as transient (process and discard). This precludes reprocessing past data to support newer versions of the stream processing applications.

1.2 Research Questions

We formulated the following research questions to guide this study.

RQ-1: How can we develop a holistic methodology to address *both* the ingestion and processing challenges pertaining to high volume data streams originating in CSEs?

RQ-2: How can we support data volume reductions in both single-feature and multi-feature streams?

RQ-3: How can we leverage the data reduction technique employed at the edges to improve performance of the stream processing applications?

RQ-4: How can we develop a data reduction technique that can accommodate future application changes by seamlessly integrating with existing data infrastructures?

1.3 Approach Summary

Our methodology is centered around the idea of *sketched streams*. Our reference implementation is code named *Pebbles*. An observational stream is partitioned into contiguous, non-overlapping temporal windows called *segments* and observations within each of the segments are represented using a sketch — a space efficient representation of the multi-feature observations. Towards this end, as part of this study we present a new sketching algorithm, referred to as *Pebbles sketching algorithm*, designed especially for time-series data streams. Pebbles sketching algorithm is different from other frequency-based sketch algorithms such as Augmented Sketch [33], Count-Min [34], and Misra-Gries [35], because it can preserve the ordering between observations within a segment, which is critical for most stream processing use cases. Existing frequency-based sketch algorithms offer point queries limited to cardinality or occurrence frequencies [36]. In addition to the ability to reconstruct the ordered stream segments from the sketch and supporting frequency queries, Pebbles sketches natively support certain stream processing related operations on stream segments such as partitioning and transformations.

We leverage processing capabilities at the edges of the network to preprocess streams to produce *sketched streams*. As we demonstrate in our benchmarks, this process reduces the volume of the observational streams significantly while reducing the energy footprint at the edge devices. In Pebbles sketching algorithm, we compromise the resolution of individual feature values, while preserving how the feature values vary over time as well as with respect to other features through a process called *discretization*. Our adaptive discretization algorithm guarantees that the error introduced by discretization is always below the preconfigured threshold despite concept drifts in the data stream as well as anomalies. A Pebbles sketch maintains the discretized observations using a set of bitmaps. Due to gradually evolving nature of most streams, consecutive observations map to a single discretized observation resulting in sparse bitmaps which are amenable for further compaction.

Once the sketched streams are ingested into the center, they can be processed through the Pebbles sketch-aware stream processing API to achieve high performance. The Pebbles stream processing API is a drop-in extension to standard stream processing APIs without requiring changes to the stream processing infrastructure or any of the existing applications. Through its lower memory and network footprint, sketched streams are able to improve throughput, bandwidth utilization, context-switching, and garbage collection overheads (Section 3.3). Sketches can be converted back to its constituent observations within a stream processing operator through a process called *materialization*. Pebbles sketches support different materialization modes such as *all*, *sample*, and *topK/bottomK*; applications can switch between processing modes dynamically based on workload and system conditions. Further, we support in-sketch operations where certain operations can be performed without materializing the sketch, therefore yielding higher throughputs and reduced memory utilization compared to their standard implementations. While we focus on centralized processing in this study, sketched streams are an effective construct for stream processing at resource constrained edge devices due to its compact network and memory footprints, efficient in-sketch operations, and integration with existing stream processing APIs.

1.4 Paper Contributions

- A new sketching algorithm designed from the ground up for space-efficient, order-preserving representations of multi-feature, time-series data streams with guaranteed accuracy
- A novel methodology based on data sketching to address both ingestion and processing challenges pertaining to data streams originating in CSE settings
- High-performance, sketch-aware stream processing API that can be seamlessly integrated with existing data infrastructures, both centralized and federated, used by organizations

We validate the proposed methodology using three different real-world streaming datasets from diverse domains such as industrial monitoring, smart homes, and atmospheric monitoring. Our empirical benchmarks show a $\sim$7.3$\times$ and $\sim$8$\times$ improvement in energy consumption and data transfer at the edges respectively. Pebbles sketch-aware stream processing API improves throughput up to $\sim$27$\times$ compared to traditional stream processing APIs while substantially improving bandwidth utilization and reducing the execution and memory management overhead.

Paper Organization: The remainder of the paper is organized as follows. In Section 2, we outline the overall system architecture. We discuss our methodology in Section 3, followed by the system benchmarks in Section 4. Related work is discussed in Section 5. We present conclusions and future work in Section 6.

2 System Architecture

In this section, we introduce the key components and their interactions in our systems architecture, code named *Pebbles*. This is depicted in Figure 1.

The **Pebbles edge module** is deployed in proximity to the source of the data stream. Edge modules are responsible for

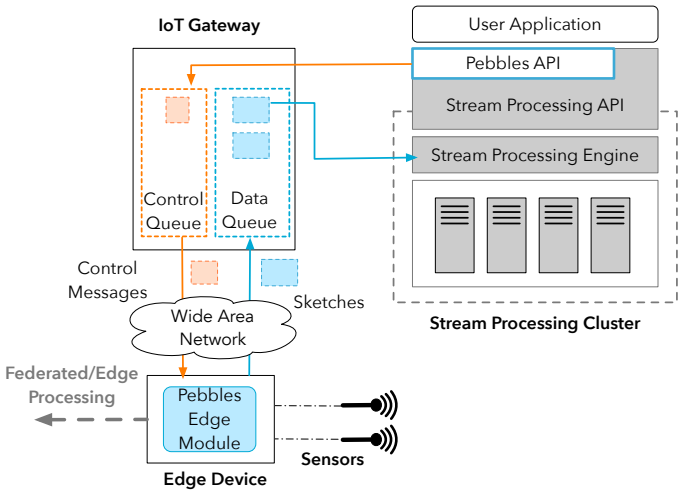


Fig. 1: System Architecture of Pebbles. Pebbles edge module is responsible for generating sketches and send them over to the center via the IoT gateway. Pebbles center module extends the API of the stream processing engines allowing users to write applications on sketched data streams.

converting the observational streams into sketched streams. Data from sensors can be fetched into the edge module using either push or pull ingestion modes. For instance, a data collector node of a sensor network can be used to deploy the edge module which contacts the individual sensors at regular intervals and construct a multi-feature observation [10]. Cloudlets [19], fog computing devices [21], mobile phones, and telco clouds at the edges [20] are other possible target devices to run Pebbles edge modules. Alternatively, the methodology of the edge module we explain in the following sections can be integrated into various edge computing modules such as Amazon's Greengrass [37] and Apache Edgent [38]. The sketches generated at the end of each segment is transferred to the processing layer (e.g.,: IoT gateway) via MQTT [39] or TCP. MQTT is a lightweight machine-to-machine (M2M) protocol built on top of TCP/IP to be used devices with low power profiles and unreliable networks.

The **IoT gateway** acts as the intermediary to provide space and time decoupling between the edge modules and the data processing middleware (in center). Edge modules can join and leave the data ingestion pipeline without any changes in the user applications running at the center. The stream processing cluster can be subjected to horizontal scaling, node failures, etc. without requiring any changes to data sink configurations used by the edge modules. This design also facilitates aggregating sub-streams from multiple, spatially distributed data sources into a single data stream. Further, the IoT gateway is used to establish a control channel from the center to the edge modules. IoT gateways are a common component in IoT deployments [40], [9] and usually implemented using a message broker. In our implementation, we used an Apache Kafka [41] to implement the IoT gateway. Using a message broker supporting multiple consumers like Kafka enables other data consuming applications within the organization (e.g.: batch jobs) in addition to the stream processing jobs to consume the data streams without having to maintain multiple copies of data. In a real world deployment, it is possible to use the existing message brokers in the data ingestion infrastructure to implement the IoT gateway without any additional operational cost.

Pebbles stream processing module extends the API of the stream processing engine to handle sketched data streams. It acts as a facade for the sketched data streams by internally handling the materialization of sketches and in-sketch operations transparent to the user. Currently, Pebbles support Apache Storm [42] and Neptune [43] stream processing engines. Data ingestion operators in stream processing applications (e.g.,: Spouts in Apache Storm) subscribe to the topics in the IoT gateway corresponding to the input data streams. Optionally, user applications can send control messages using the Pebbles API to the edge modules contributing to a particular data stream enabling server-initiated steering. Our design does not preclude the use of Pebbles stream processing module at the edges of the network.

3 Methodology

We present our methodology based on sketched streams for processing voluminous, high-velocity data streams generated at the edges of the network. We leverage the power of edge devices to preprocess data to effectively reduce the volume and the cloud to provide scalable streaming analytics over the data streams. We also list the research questions addressed by each subsection.

3.1 Preprocessing at the Edges (RQ-1, RQ-2)

The Pebbles edge module is responsible for converting a multi-feature observational stream into a stream of sketches. An observational stream is partitioned into non-overlapping windows, called *segments*, based on a preconfigured window length. Segments are similar to tumbling windows as defined in stream processing literature [17]. Observations within a segment is represented using a sketch instance. Figure 2 depicts the key components of the Pebbles edge module.

Once a multi-feature observation is made available to the Pebbles edge module, it is transformed into a *feature-bin combination (FBC)*. The resulting feature-bin combination is then included in the sketch corresponding to the current time segment by the *Sketch Manager*. The sketch Manager tracks the progress of the stream and splits it at the correct boundaries generating new segments. Once a new segment is instantiated, the sketch and the metadata corresponding to the previous segment is serialized and passed to the *Transport Handler*. The transport Handler is responsible for handling communications

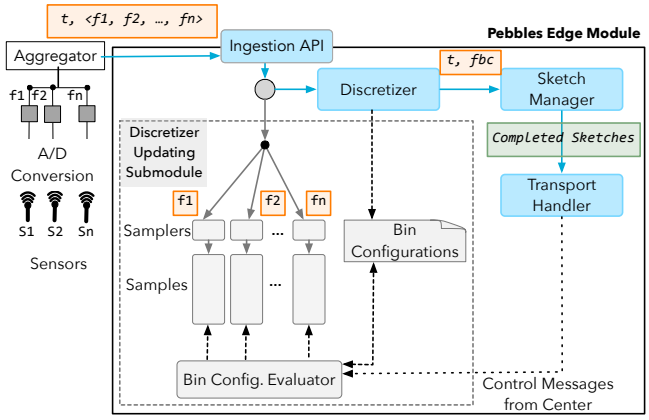


Fig. 2: Components of the Pebbles edge module. Multi-feature observations are discretized and sketched prior to sending to the center. Bin configurations are dynamically updated as the stream evolves or upon the request from the user applications.

with the center via the IoT gateway via the configured transport protocol. Next, we will discuss the discretization and sketching phases in detail.

3.1.1 Discretization and Generating FBCs

Discretization is the process of discretizing continuous, individual feature values by mapping them to corresponding bins. To aid this process, the Pebbles edge module maintains a bin configuration for each feature in an observational stream — a set of points which partitions the range of the possible values of the feature into a given number of bins. During discretization, each feature value in an observation is mapped to the appropriate bin in its bin configuration. The identifiers of the corresponding bins are concatenated together to construct the feature-bin combination corresponding to the observation. For instance, let's consider a simple stream with two features with the bin configurations: {[100.0, 120.9), [120.9, 150.1), [150.1, 200)} and {[-0.1, -0.02), [-0.02, 0.05), [0.05, 1.1), [1.1, 1.9)}. Suppose the observation at time t' is $\langle t',\ \langle 129.1, 0.09\rangle\rangle$. The first feature value 129.1 is mapped to the second bin, hence replaced by identifier 2. Similarly the second feature value is mapped to the third bin (with the identifier 3) resulting in a feature-bin combination of $\langle 2,\ 3\rangle$. It should be noted that our methodology is equally applicable for single-feature streams as well in which case a FBC will contain a single bin identifier.

Discretization exploits the gradually evolving nature of feature streams to achieve compression similar to existing edge data reduction schemes [10], [27], [26]. Multiple subsequent feature values are likely to be resolved into the same bin, which eventually results in a few feature-bin combinations representing all the observations within a segment. We leverage this behavior within our sketching algorithm to effectively reduce the volumes of the data streams.

Estimating bin configurations: We leverage Online Kernel Density Estimation (oKDE) [44], [45] to autonomously generate bin configurations — both the number of bins and the range of values encapsulated by individual bins. For each feature, the probability density function is estimated using online kernel density estimation with respect to a sample of observed values. The numerical range of feature values is then partitioned into a given number of bins such that each bin having an equal area under the curve. This results in a bin configuration with each bin having an equal probability of landing the next observation. An example probability density function and the associated bin configuration is depicted in the first sub-figure of Figure 3. Discretization is a non-reversible operation, therefore reconstructing the feature-value vectors from the feature-bin combinations incurs an estimation error, called the *discretization error*. In our reference implementation the middle value of a bin is considered the estimated value for the bin introducing a maximum error of half the bin width. Using an oKDE based binning configuration ensures that the high-density bins (bins with high probability) cause a lower discretization error. The number of bins in a bin configuration is the minimum number of bins that satisfy the desired discretization error threshold with respect to an online updated sample of observed values. The discretization error of a sample can be quantified using various metrics such as descriptive statistics and statistical distance. A key requirement is the ability to calculate this metric in near real time on resource-constrained edge devices. For instance, in our benchmarks we used normalized root means square error (NRMSE) to measure the discretization error and set the threshold to 2.5% for each feature.

Enforcing an upper-bound on discretization error: We extend the basic discretization algorithm outlined in Section 3.1.1 to ensure that the discretization error for each feature does not exceed the preconfigured threshold. The basic discretization algorithm is susceptible to higher discretization errors in following scenarios.

- Anomalies - Anomalies usually fall into low-density bins (with low probability) therefore increasing the discretization error.
- Concept drift - As a stream evolves, the probability density function estimated by the oKDE may not adequately represent the current state of the stream. This may result in more observations being discretized into low-density bins.

Our improved discretization algorithm, *adaptive discretization*, provides a guaranteed upper bound on the discretization error by only discretizing an observed value if the discretization error is less than the threshold. Otherwise, the algorithm sends the observed value *as is* resulting in a discretization error of 0.0%. In the latter case, as an optimization we also inject a new bin into the bin configuration with the observed value as the middle value of the bin — we use the bin identifier of the newly introduced bin to construct the feature-bin combination. This optimization increases the likelihood of subsequent observations getting discretized using the newly introduced bin in the case of a concept drift. Discretized regions corresponding to a probability density function and the associated bin configuration is illustrated in Figure 3. Another advantage of our adaptive discretization is the ability to preserve anomalies. Adaptive discretization prevents high-error discretization of anomalous feature values which is useful for anomaly detection use-cases; this is illustrated in Figure 4.

The adaptive discretization algorithm is outlined in Algorithm 1. The time complexity of discretizing an observation is $O(mlogk)$ where m is the number of features and k is the average number of bins in a bin configuration. The average number of bins is usually in the order of hundreds based on our empirical observations.

Algorithm 1 Adaptive discretization algorithm

v_i : Value of feature i at a given time
bc_i : Bin configuration for feature i
f_{min}, f_{max} : Min and Max values for feature i
th_i : Discretization error threshold for feature i

function ADAPTIVE_DISCRETIZE($v_i, bc_i, f_{min}, f_{max}, th_i$)
  $b_k \leftarrow highest\_bin\_lower\_than\_value(bc_i, v_i)$
  $b_{k+1} \leftarrow next\_bin(bc_i, b_k)$
  $midpoint \leftarrow b_k + (b_{k+1} - b_k)/2$
  $nrmse \leftarrow RMSE(midpoint, v_i)/(f_{max} - f_{min})$
  if $nrmse < th_i$ **then return** b_k
  else
    if $v_i < midpoint$ **then**
      $b' = b_k + 2 * (v_i - b_k)$
    else
      $b' = b_{k+1} - 2 * (b_{k+1} - v_i)$
    end if
    $add\_new\_bin(bc_i, b')$ **return** b'
  end if
end function

Pebbles dynamically updates the bin configurations over time to mitigate concept drifts. Otherwise, the size of a bin configuration can increase due to injected bins causing issues such as insufficient memory to hold bin configurations at edge devices, higher discretization costs, and higher network foot-

prints after serializing sketches (higher bin counts require more bits to represent a bin identifier when constructing feature-bin combinations). These updates are performed independently per feature in a particular stream. We evaluate the existing bin configuration without the injected bins against an online updated sample for each feature. If the overall discretization error for the sample is higher than the threshold, the system triggers a bin configuration update in the background. In order to capture concept drift, the sample should be representative of the recently observed values of a particular feature. Reservoir sampling [46] is a single-pass unbiased sampling algorithm designed for inputs with unknown sizes (such as streams) and ensures that every element has the same probability of getting selected — sampled elements are uniformly distributed throughput the stream. Weighted reservoir sampling extends this behavior through a weight function to assign different selection probabilities for the sample.

We support two weighted reservoir sampling algorithms with temporal bias functions that assign more weight to recent items in the stream: Aggarwal's reservoir sampling algorithm [47] and Uniform Variable Input Rate Biased Sampling [48]. While both algorithms are capable of maintaining samples that are well representative of the recent items in a stream, Uniform Variable Input Rate Biased Sampling is able to deal with variable observation rates of streams. Updating samples, evaluating current bin configuration, and updating bin configurations are performed in the background in parallel to the data ingestion for each feature separately. This allows the discretization process to autonomously be in sync with the evolution of individual features.

Figure 5 depicts a microbenchmark that demonstrates the effectiveness of different sampling algorithms on evaluating the effectiveness of a bin configuration. We recorded two metrics: (1) actual error - discretization error of the stream data over a sliding window of length 2 seconds (with 1 second sliding period) was used for the evaluation (2) observed error - discretization error of the sample at a given time. If the observed

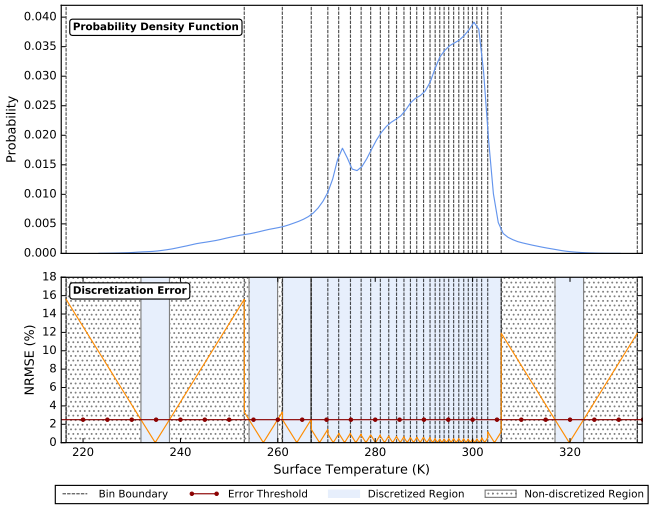


Fig. 3: Probability density function (PDF) and the associated bin configuration for surface temperature feature in the NOAA North American Mesoscale forecast dataset at the beginning of the stream. Our discretization scheme enforces an upper-bound on the discretization error by selectively discretizing feature values that result in a lower discretization error. The PDF is updated in the background and the bins are recalculated as the stream evolves.

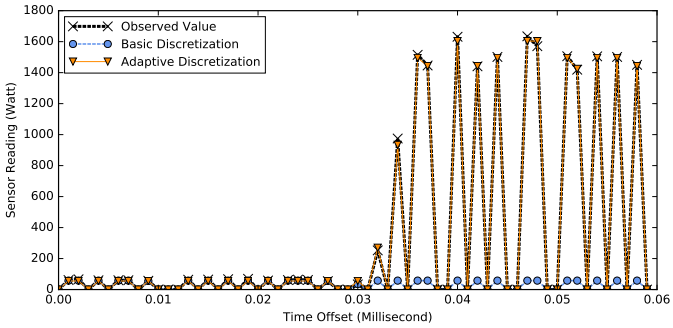


Fig. 4: Adaptive discretization provides a guaranteed upper-bound for the discretization error while representing anomalies with a lower error compared to the basic discretization algorithm. This figure demonstrates how the adaptive discretization algorithm is able to capture a short-term peak in the load of one of the smart plugs in the smart home dataset.

discretization error exceeds the preconfigured threshold (2.5% in this benchmark), an update to the bin configuration is triggered. As can be observed in Figure 5, observed errors for both weighted reservoir sampling techniques following the actual error closely; therefore, triggering bin configuration updates at appropriate intervals compared to the traditional reservoir sampling.

We also support serverside-steering for the discretization process. Stream processing applications can trigger a bin configuration update through control messages as shown in Figure 2. Either they can adjust the discretization error threshold or

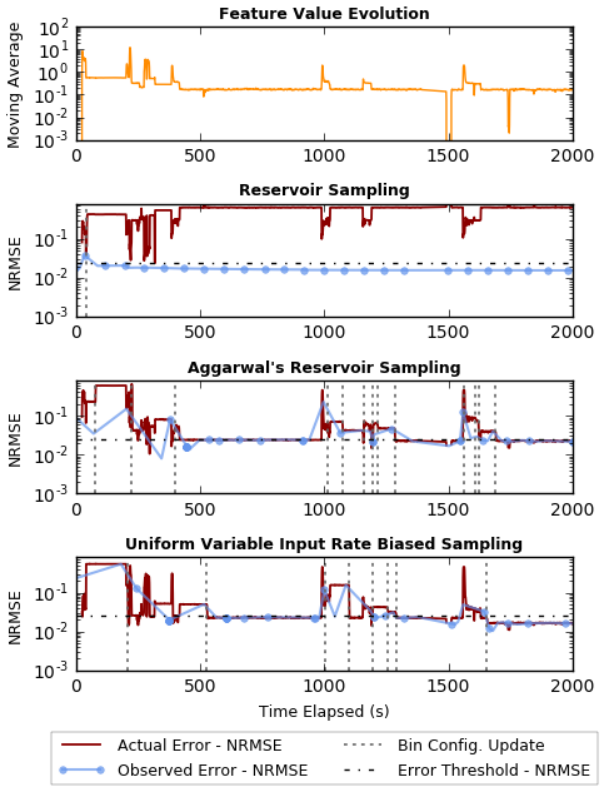


Fig. 5: Effectiveness of weighted reservoir sampling when assessing bin configurations. Weighted reservoir algorithms can effectively capture the concept drift when compared to the reservoir sampling algorithm.

directly override the bin configuration of a particular feature with a custom bin configuration. Additional fine tuning of the discretization process such as adjusting the frequency of periodic bin configuration evaluation and enabling/disabling online updates to the bin configuration are supported. The resulting end-to-end dynamism of the data ingestion pipeline facilitates accommodating changes at the stream processing system such as addition and removal of new data processing jobs.

The bin configuration used for discretization should be available at the stream processing layer in order to estimate the individual feature values from the bin identifiers in the FBCs. We piggyback on the actual messages (with serialized sketches) to the send the bin configuration updates instead of sending them over a separate stream. Otherwise a stream join is required at the center between the data stream and the bin configuration update stream to determine the corresponding bin configuration update required to resolve the FBCs. A higher QoS is used for guaranteed delivery of the messages containing the bin configuration updates. For each stream, a monotonically increasing version number is assigned to a set of bin configuration updates performed within a time segment. Every sketch carries the version number of the bin configuration used for discretization when it was updated. This version number helps with dealing with out-of-order arrivals of messages, which are common especially when messages are transferred over public networks [49]. In case of an out-of-order arrivals, messages will temporarily reside in a memory buffer until the message with appropriate bin configuration update arrives. Most modern stream processing engines are equipped with built-in buffers to deal with such short-term, out-of-order arrivals.

3.1.2 Pebbles Sketching Algorithm (RQ-2)

The Pebbles sketching algorithm is an *order preserving* sketching algorithm for *multi-feature streams* designed from the ground up. It addresses a common limitation of current frequency based sketching algorithms: the inability to preserve the ordering between observations. Maintaining the ordering between observations is critical for most stream processing use cases. For instance, maintaining temporal windows and tracking state changes over time require the ordering between observations.

Let's assume a data stream (already discretized into feature bin combinations) with observations produced at every p time units, i.e., with a frequency of $1/p$. The m observations produced within the time segment $[t_n, t_{n+mp})$ in stream S is denoted as;

$$S_{[t_n,t_{n+mp})} = \{(t_n, fbc_n), (t_{n+p}, fbc_{n+p}), ..., (t_{n+(m-1)p}, fbc_{n+(m-1)p})\}$$

where $fbc_i : n \leq i < n+mp$ is a feature bin combination occurring at time i. By encoding the timestamp t_i of an observation as a temporal offset o_i from the starting timestamp of the time segment t_n as a multiple of p, the same set of observations can be represented as;

$$S_{[t_n,t_{n+mp})} = t_n, \{o_i, f_i \mid o_i = (t_i - t_n)/p;\ 0 \leq i < m)\}$$

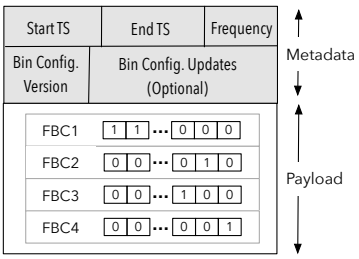


Fig. 6: The structure of a Pebbles sketch instance. Metadata encapsulates information needed for materialization. Bitmaps in the payload are dynamically compressed.

A Pebbles sketch maintains an inverted index of unique feature-bin combinations observed within a time segment as a set of ⟨*key, value*⟩ pairs where the set of keys corresponds to the set of unique feature-bin combinations. The value is the list of temporal offsets relative to the segment start timestamp (as calculated above) in ascending order where the given feature-bin combination occurred. For instance, let's consider a time segment [10,20) with an observation period of 2, therefore generating 5 observations — `(10, fbc1), (12, fbc1), (14, fbc2), (16, fbc1), (18, fbc1)`. The corresponding sketch instance will store ⟨`fbc1, {0,1,3,4}`⟩ and ⟨`fbc2, {2}`⟩.

Pebbles sketching algorithm is a linear sketching algorithm where a linear transform over the input stream segments are applied to generate a sketch instance. In a linear sketch, an update on the sketch has the same impact irrespective of the previous updates [34]. The sketch of two adjacent segments can be calculated by merging sketches for the individual segments.

Maintaining an inverted index of temporal offsets can still incur a significant space overhead. We exploit the gradually evolving nature of most observational streams to gain further space reductions. Due to this gradually evolving nature, we posit that the consecutive observations are likely to transformed into a single feature bin combination once discretized. This results in stretches of consecutive temporal offsets within the inverted indexes making them amenable to compression using techniques such as compressed bitmaps and inverted lists.

When using compressed bitmaps, each list of offsets within the inverted index is represented using a bitmap. The number of bits of the bitmap is equal to the number of observations within a segment — the position of a bit within a bitmap represents the temporal offset as a multiple of inter-observation interval. If a certain feature bin combination occurs, the corresponding bitmap is located within the inverted index and the bit in the position equal to the temporal offset is set. Following the example used before, the bitmap representation of the Pebbles sktech instance will be ⟨`fbc1, {11011}`⟩ and ⟨`fbc2, {00100}`⟩. Bitmap compression is a well studied research area with several contributions such as WAH [50], EWAH [51], CONCISE [52], and Roaring [53]. WAH, EWAH, and CONCISE are variants of run-length encoding where series of identical bits are compressed to the bit value and the count. Roaring adapts a hybrid compression technique incorporating an uncompressed integer list and uncompressed bitmap. Inverted lists are an alternative to compressed bitmaps which usually compresses the differences (a.k.a *deltas*) between the successive integers [54] — e.g., Variable Byte [55], PforDelta [56], and Simple9 [57]. In addition to being highly compressible, bitmaps support efficient bit-wise operations that are useful for high-performance data manipulations as discussed in Section 3.3.2.

Figure 6 illustrates the key building blocks of a Pebbles sketch. The metadata includes the temporal bounds of the segments and the frequency which are required for reconstructing the actual timestamps from the temporal offsets during the materialization. Additionally, the start timestamp provides ordering between messages within a stream. These metadata fields are included in every sketch instance to accommodate dynamic changes in sensing configurations such as frequency and segment duration. Version number of the bin configuration

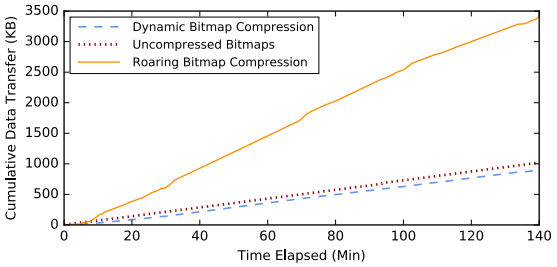


Fig. 7: Effectiveness of dynamic compression in the reference implementation of the Pebbles sketch algorithm.

used for discretization, and any updates to the bin configurations are required for approximating the observations back from the FBCs.

For our reference implementation of Pebbles sketching algorithm, we used a combination of uncompressed bitmaps and Roaring bitmaps (paired with run-length encoding) to implement the inverted indexes. We chose Roaring bitmaps over other bitmap compression techniques for:(1) space efficiency - desirable at resource constrained edge devices, and (2) faster decompression during materialization [54].

Our compression scheme is dynamic. None of the inverted index compression techniques are effective with highly randomized sequences of integers — in fact, they incur more overhead than maintaining an uncompressed bitmap. We initialize a Pebbles sketch instance with uncompressed bitmaps and maintain an online updated metric between the ratio of the number of bits set and the highest offset for each bitmap. At the end of the segment, if the ratio of these metrics is too high or too low, it is an indication that the bitmap is mostly empty or full. In such cases, we convert the uncompressed bitmap into a roaring bitmap and further optimize it with run-length encoding; otherwise we continue to use the uncompressed bitmap as is. In Figure 7, we demonstrate the effectiveness of the dynamic compression algorithm. We used the feature with the highest variability in Smart Home dataset [58] in this microbenchmark. Enabling roaring bitmaps by default is not effective due to the randomness of the data. Dynamically enabling the compression based on the occupancy heuristic can reduce the data transfer by ∼74% compared to using roaring-bitmaps and by ∼12% compared to using regular bitmaps alone.

The definition of Pebbles sketch algorithm does not preclude different implementations optimized for the use case in hand depending on the characteristics of the datasets and types of operations performed at the center.

3.1.3 Trade-off Space Analysis

In order to achieve reductions in data volumes, we trade off resolution of the individual feature values and latency. A sketch is not transferred until the segment elapses — providing latency characteristics similar to those in micro-batching systems [59] where a stream is split along the temporal axis and transferred in bundled micro-batches. There is a trade-off between the latency introduced by segmentation and the compaction achieved by sketching segment duration in Pebbles is configurable and can be even dynamically controlled allowing users to select an appropriate segment duration based on latency requirements. Our methodology is applicable if the features of a data stream can be represented as numerical values and the applications can tolerate a controlled loss of precision of individual feature values and the increased end-to-end latency introduced due to segmentation.

3.2 Transferring Data to the Center (RQ-4)

Once a segment expires, a sketch is transferred to the IoT gateway. Sketched streams reduces both the number of communications initiated as well as the overall volume of the data transferred. As we show in our benchmarks, the overhead of the sketch generation is outweighed by the energy saving due to reduced communication with the center. Also, this approach ensures that the limited bandwidth available at the edge devices are utilized more efficiently.

Sketched data streams are published over dedicated topics assigned for each stream within the IoT gateway similar to most regular data stream ingestion pipelines. Stream processing applications subscribe to the topics corresponding to their input streams. Given that the bin configuration updates are available as part of the metadata, if a new stream processing application is deployed it needs to consume the entire set of previous messages containing bin configuration updates in order to be able to construct an up to date bin configuration. This requires storage as well as processing of the entire stream which is expensive. There are two solutions to address this issue:

- Maintaining the history of bin configuration updates as shared state across all applications — e.g.: using an in-memory key-value store.
- Storing another copy of messages with bin configuration updates on a separate topic. New applications will consume the messages in this topic up to the offset where they want to start consuming the data stream.

In our reference implementation, we opted for the second option because it does not incur the operational overhead of another system. Also the absence of a centralized metadata system makes the integration of Pebbles into edge/federated processing use cases easier. Bin configuration updates are typically manageable when compared to the total number of messages and can be subjected to merging if required — an older update can be replaced by a newer update to the bin configuration of a particular feature and bin updates to multiple features can be merged into a single message.

Due to the smaller storage footprints for sketched streams, they can be stored for extended periods of time in the IoT gateway. This enables reprocessing of the streams in case of any updates to the stream processing applications due to software bugs uncovered later or changed business requirements. Kappa architecture [18] is one architectural pattern to support processing of past streams by storing them in message broker infrastructures (similar to the IoT gateway), and processing them with higher parallelism.

3.3 Stream Processing API (RQ-1, RQ-3, RQ-4)

A typical stream processing API allows stream processing applications to be designed as directed acyclic graphs (DAG) comprising of stream operators (vertices) connected via streams (edges). An operator can be either an ingestion operator which ingests the input streams from the external sources into the DAG or stream processors which carries out part of the application logic and may produce a stream of messages to the downstream processors (derived streams). The Pebbles stream processing API follows the same model, but considers a different unit of workload — sketch instances for processing and communication between stream operators in the DAG.

Given that a Pebbles sketch instance is a compact representation of a set of observations from input streams or messages produced as part of the stream processor logic; the Pebbles stream processing API achieves the same benefits as *batching enabled stream processing*. In batching enabled stream processing,

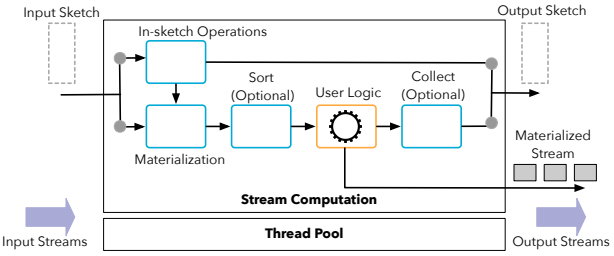


Fig. 8: Implementation of the Pebbles sketch processing API

a group of messages is considered as the unit of data for transferring and processing in order to improve the throughput by amortizing the communication and stream processor execution costs [60]. Processor execution costs are improved through amortizing the costs associated with loading the instruction cache, scheduling, and context switches over a group of messages. Transferring larger payloads compared to smaller payloads improves the efficiency of the network utilization eventually leading to improved throughputs. Sketched streams are more compact than batched streams packing more observations in a single sketch instance through controlled trade-off of accuracy. This further improves processing throughput, memory overhead, as well as per-message execution overhead as we demonstrate in our systems benchmarks. Further, Pebbles sketches can support in-sketch operations (as explained below) where certain operations can be performed without expanding a sketch into the set of constituent observations delivering high-throughputs and reduced memory overhead.

The Pebbles stream processing implementation is designed as an extension to the existing stream processing APIs. Currently we support Apache Storm [42] and Neptune [43]. It is deployed as a drop-in library similar to any runtime dependency without requiring any modifications to the stream processing engine itself. This is advantageous because:

- The Pebbles stream processing applications can execute alongside stream processing applications developed with regular stream processing APIs
- This supports hybrid stream processing applications - Applications where a subset of the stream processors are implemented using Pebbles API while the remainder are implemented using the regular stream processing API
- It builds on the large body of existing work in the development of feature-rich, robust stream processing engines

In this paper, we have used stream processing engines to prototype the Pebbles stream processing API. But our proposed methodology is applicable for micro-batching systems like Spark Streaming [59] as well. We will discuss the key concepts of the Pebbles stream processing API next.

3.3.1 Sketch Materialization

Materialization is the process of converting a sketch back to the set of individual discretized observations. Pebbles provides different materialization modes: ① **all:** Generate *all* discretized observations represented by a sketch instance, ② **sample:** Generate a *sample* of the discretized observations represented by a sketch based on a user-defined sampling function, and ③ **topK/bottomK:** Generate the observations corresponding to *most/least frequent* observations.

Materialization internally expands bitmaps of the underlying Pebbles sketch instance and creates an observation for each set bit in the bitmaps. For materialization modes such as *topK/bottomK*, only a subset of the bitmaps are expanded based on their cardinality. The timestamps of the observations are generated based on the starting timestamp of the sketch and frequency (available as part of the metadata) combined with the offset (derived from the bit position within the bitmaps). Having support for multiple materialization modes at the API level not only enables stream processors to use the appropriate materialization mode for their processing logic, but also allows dynamically adjusting their processing semantics during runtime based on metrics related to operating conditions and workload. More specifically, a stream processor can dynamically switch between different materialization modes in runtime or adjust the parameters of its current processing mode. For instance, if a processor cannot keep up with the incoming stream, it can temporarily switch from *all* mode to *sample* mode essentially creating a load shedding setup or further reduce the sampling rate if it was already using *sample* mode.

The output of the materialization operation is made available either as an Iterator or a Java Stream [61]. Support for Java streams presents a functional style API to process the sketched streams with the added advantages of lazy evaluation and parallel execution.

Ordering of observations: The resulting set of discretized observations from a materialization operation is not ordered based on the timestamps by default. If required, users may opt for sorting the observations chronologically based on the observation timestamps.

Parallel execution: A Pebbles sketch instance can be materialized using multiple threads in parallel where each bitmap or contiguous block of a bitmap is expanded in parallel. Similar to sorting, parallel materialization should be explicitly enabled. Users can request a `parallelStream` if they use Java Streams or enable parallelization and provide a parallelization factor if they access observations through an Iterator.

3.3.2 In-Sketch Operations

In-sketch operations are a class of operations that can be directly performed on a Pebbles sketch instance without requiring a priori materialization. These operations directly work on the feature-bin combinations (keys) or bitmaps (values) of a Pebbles sketch instance. Following is a list of in-sketch operations supported by Pebbles sketches.

1. Key transformations - Apply transformations on FBCs such as projections, enrichments, and cleaning.

2. Filtering - Remove a subset of the FBCs based on a user-defined filtering criteria.

3. Key based partitioning - Horizontally partition a sketch based on a user-defined grouping criteria on FBCs.

4. Temporal partitioning - Vertically partition a sketch along the temporal axis such that a partition holds observations corresponding to a contiguous period of time.

5. Round-robin partitioning - Assign observations into partitions in a round-robin fashion.

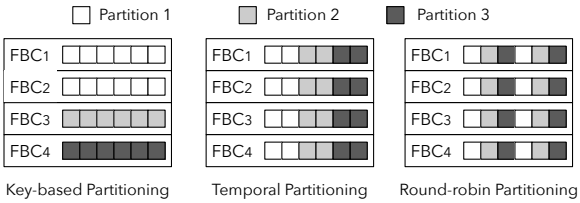


Fig. 9: Partitioning Schemes using in-sketch operations.

Figure 9 depicts various sketch partitioning schemes implemented as in-sketch operations. Given that these operations do not need to materialize the sketches in advance, they enable high-throughput processing as demonstrated in Section 4.4.2. Furthermore, they incur a less memory and garbage collection overhead by eliminating the instantiation of large number of short-lived objects. Stream processing on resource constrained devices also benefit from in-sketch operations due to these characteristics. Some in-sketch operations such as temporal partitioning benefit from the fast bit-wise operations supported by modern bitmap implementations. For instance, temporal partitioning is implemented by performing a bit-wise AND operation on the bit masks generated for each partition.

3.3.3 Collect Operation

Collect operation, part of the Pebbles' Java Stream integration, constructs a sketch back from a stream of events. A stream processor can materialize a sketch as a Java stream, run the user-defined processing logic, and pack the derived event as a sketch using the *collect* operator ready to be transmitted to the next stage. The applicability of this operator depends on the ability to represent an event as a ⟨*key, temporal offset from the base timestamp*⟩ tuple. Using sketches to transfer events within the DAG improves the overall performance of the stream processing application, especially with respect to throughput and bandwidth utilization as discussed earlier.

3.3.4 Pebbles Stream Processing Topology

Pebbles stream processing applications are directed, acyclic graphs of stream ingesters and stream processors connected via streams similar to any stream processing application. One key differences is in the Pebbles ingester, which consumes data from the IoT gateway and injects them into the remainder of the processing DAG. This ingester is responsible for resolving the feature-bin identifiers — each bin identifier is replaced by the bin boundaries based on the appropriate bin configuration. Because the bin identifier resolution is implemented as an in-sketch key-transformation operation, it does not add a significant latency or processing overhead. Once the bin identifiers are resolved, each Pebbles sketch instance becomes self-contained. If the past data need to be consumed, the ingester can be configured to consume data from the topic storing the bin configuration updates up to the required bin configuration version, and switch back to the sketched stream.

The remainder of the processing DAG can be implemented by combining the Pebbles stream API and the regular stream processing API. By maintaining the sketched streams as far as possible in the processing DAG using *in-sketch* and *collect* operations, the overall throughput of the stream processing graph can be improved substantially. The ability to use both APIs simultaneously within a single application facilitates a seamless interoperation between Pebbles and existing stream processing code. For instance, once a sketch is materialized, it can be processed as the equivalent regular data stream using the regular stream processing API.

4 Evaluation

We profiled the efficacy of our approach with respect to both data volume reduction at the edge devices and improved processing at the center using real world datasets.

4.1 Experimental Setup and Datasets

Edge devices: We used the Raspberry Pi 3 model B (1.2 Gz Quad Core Processor, 1 GB RAM) running Arch Linux and Oracle JDK 1.8.0_121 as the edge devices. Power measurements at the edge were carried out using Ubiquiti mFi mPower Mini smart plugs. MQTT was used as the messaging protocol between the edge devices and the center.

Stream Processing Cluster: We used HP DL60 servers (Intel Xeon E5-2620 2.40GHz processors, 16 GB RAM) running Fedora 28 and Oracle JDK 1.8.0_121. We used Apache Storm v1.2.2 for benchmarks involving Storm.

Datasets: We used three real world datasets in our benchmarks encompassing multiple domains: smart homes, industrial monitoring, and meteorological forecasting covering a wide range of frequencies, durations, and stream evolution patterns.

1) Smart homes dataset [58] includes readings from smart plugs deployed in set of households in Germany. Sensors attached to each smart plug periodically report current load and cumulative work since the last reset of the sensor. Each plug produces 1000 observations/s. We preprocessed the dataset to organize data based on households (entities) such that each observation contains the current load on 12 different plugs (12 features).
2) Gas sensor array under dynamic gas mixtures dataset [7], created by the BioCircuits Institute at UCSD, includes time series data from 16 chemical sensors exposed to gas mixtures at varying concentration levels. For our benchmark, we used a subset of the dataset corresponding to Ethylene and CO gas mixture containing 18 features. The frequency of the dataset is 100 observations/s.
3) NOAA North American Mesoscale forecast dataset [2] contains periodic recordings of meteorological features by weather stations deployed across north America. We considered 10 features including temperature, humidity, and precipitation for year 2014. We used an imputed version of the dataset with a frequency of 1 observations/s.

Given that streaming workloads are unbounded in real world settings, we performed our benchmarks based on the duration of our datasets. Each stream was ingested using the original frequency associated with each dataset; smart homes, gas sensor array, and NOAA datasets required ∼43 mins, 6 hours, and 24 hours respectively for complete ingestion.

4.2 Efficacy of Pebbles at the Edge (RQ-1, RQ-2)

4.2.1 Applicability to Single-feature Streams

Even though, Pebbles is designed for multi-feature streams, it is still applicable in single-feature stream settings. In this benchmark, we evaluate the efficacy of Pebbles at the edge when used with single-feature streams based on two metrics: amount of data transferred and energy consumption at the edge. We compare Pebbles with two other schemes specifically designed for single-feature streams in CSE settings.

AdaM: AdaM [27] is an adaptive monitoring framework designed for IoT devices with the goal of dynamically adjusting the sampling intensity based on the evolution of the data stream. During the stable phases of the stream, the sampling rate is lowered to reduce the data transfer and energy consumption at the edge devices. The sampling period is adjusted based on a probabilistic metric calculated based on the deviation between the actual and estimated standard deviation (future observations are estimated using a probabilistic exponential moving average model).

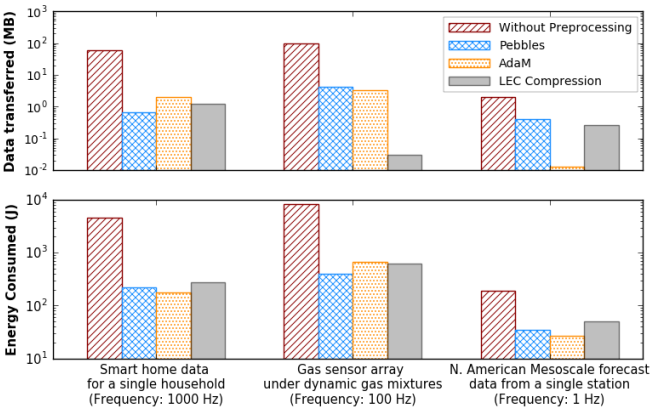


Fig. 10: Effectiveness of Pebbles sketch algorithm in comparison to Lossless Entropy Compression (LEC) and Adaptive Monitoring Framework (AdaM) in **single-feature stream** settings.

LEC: Lossless Entropy Encoding [10] (LEC) is a lookup table based lightweight compression algorithm proposed for wireless sensor networks. A set of observations are modeled as a series of differences by taking the difference between each observation and its predecessor. LEC operates on the assumption that consecutive readings do not deviate much from each other, hence their differences are smaller. Each difference is then encoded using a bit string which is a concatenation between the number of bits required to represent the difference (looked up from a dictionary of Huffman codes) and the actual difference. Huffman codes ensure that frequent values are represented using shorter bit sequences.

For this benchmark, data from an individual entity was considered for each of the datasets to simulate a stream originating at an edge device. We have chosen the feature with highest variability in each of the datasets. AdaM was configured to provide the approximately the same accuracy as Pebbles (NRMSE = 2.5%). For LEC, consecutive observations were batched and compressed; the batching interval was set equal to the segment length used by Pebbles. The energy consumption encompasses the preprocessing and data transfer over MQTT.

Figure 10 summarizes the results of this benchmark. **Pebbles demonstrated significant improvements in data transfer (by ∼5 - 91.3×) and energy consumption (by ∼5.4 - 20.6×) across all three datasets.** In general, Pebbles' performance was comparable to the other two schemes. It should be noted that Pebbles is applicable for both multi-feature and single-feature data streams whereas the other schemes are limited to single-feature streams. We expect to see similar improvements as the cumulative ingested data volumes grow with both multiple edge devices and time.

4.2.2 *Applicability to Multi-feature Streams*

In this benchmark, we compared the effectiveness of Pebbles at the edge with multi-feature streams. Similar to Section 4.2.1, we have used data from a single entity but considered multiple features instead of a single feature. We compared Pebbles with binary compression using the following metrics: amount of data transfer and energy consumption. LZ4 [62] was chosen as the binary compression algorithm because of its ability to offer a good balance between the compression ratio and speed and configurable compression levels. LZ4 was evaluated under two settings: high compression (provides highest possible compression in the expense of compression speed and energy) and fast compression (provides faster and energy efficient compression by compromising the compression ratio). Observations occurring during a time interval equal to the length of a time segment were batched together for compression.

As listed in Figure 11, **Pebbles was able achieve significant improvements in data transfer (by ∼1.5 - 8.1×) and energy consumption (by ∼3 - 7.3×).** Pebbles was able demonstrated a comparable performance with respect to data transfer and energy consumption compared to both configurations of LZ4. LZ4 fast compression demonstrated the lowest energy consumption compared to other schemes. Pebbles not only improves the bandwidth utilization and energy consumption at the edges but also enables efficient processing at the center using its sketch-aware Stream Processing API. A disadvantage of binary compression is that the data needs to be decompressed at the stream operators contributing to additional processing overheads.

4.2.3 *Evaluating the Discretization Error*

We quantified the discretization error observed by individual features within each dataset as summarized using a histogram in Figure 12. The upper-bound on the discretization error is set to 2.5%. Pebbles discretization algorithm was able to maintain the discretization error well below the given threshold.

4.3 Trade-off Analysis: Discretization Error, Data Transfer, and Energy Consumption

We analyzed how the amount of data transfer and energy consumed at the edges change with different discretization error thresholds. We used the smart home dataset (all 12 features), and each experiment was launched with the same base bin configuration. As depicted in Figure 13, with the discretization error threshold increasing (after an error threshold of 5% NRMSE), fewer new bins are introduced by the adaptive discretization algorithm. This causes the gains from sketched streams generated at higher error thresholds to plateau resulting in slower growth in energy reduction and data transfer volumes. Furthermore, the high correlation between the amount of data transferred and the energy consumption also validates that communication is the dominant energy consumption factor at edge devices as verified by previous research [14], [10], [15].

4.4 Pebbles Stream Processing API (RQ-1, RQ-3)

4.4.1 *Improving Performance at the Center*

Figure 14 depicts the performance improvements delivered by Pebbles stream processing APIs of Neptune [43] and Apache

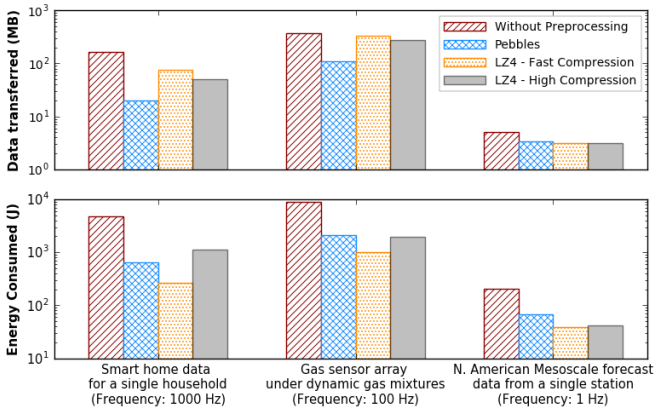


Fig. 11: Effectiveness of Pebbles at the edge w.r.t amount of data transferred and energy consumption in **multi-feature stream** settings compared to binary compression using LZ4 algorithm.

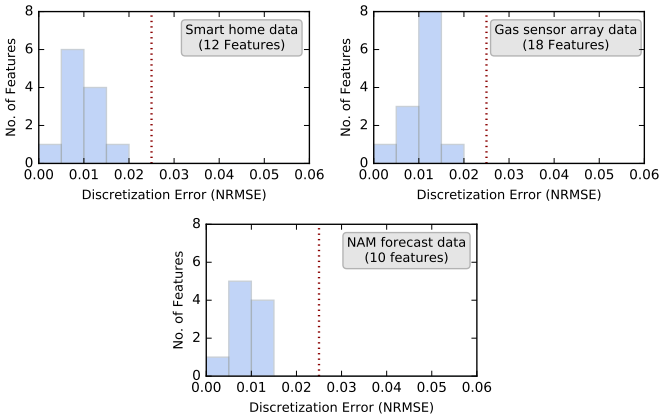


Fig. 12: Histogram depicting the discretization error observed for individual features measured as NRMSE. Pebbles is able to maintain the discretization error below a threshold of 2.5%.

Storm [42] compared to the regular stream processing APIs. We evaluated regular stream processing APIs under two configurations: one-message-at-a-time and batched streaming. We used a data processing DAG with two vertices: an ingester and a processor. The processor is responsible for calculating the moving average over a sliding window for each feature in the smart plug dataset. Sketches are materialized (using the default serial mode) with sorting enabled before processing.

Pebbles outperforms the native one-message-at-a-time processing mode w.r.t. all metrics. For throughput, bandwidth utilization, and voluntary context-switches, Pebbles outperforms batched streams due to its compactness and the high transfer rate. Ingester nodes are able to transfer more sketches per unit time compared to the number of batches due to less serialization overhead. **Pebbles' Neptune API can achieve ∼27× and ∼7× improvement in throughput compared to one-message-at-a-time and batched streaming modes respectively.** Both Pebbles and batched streaming incur similar garbage collection overhead which is significantly lower than the one-message-at-a-time configuration. This is due to the significantly lower short-lived objects being created at the transport layer when multiple messages are transferred as a single unit. Finally, we measured how well each scheme can tolerate transient performance degradations that can occur due to varying workload and system conditions [63]. If a stream processor cannot keep up with the workload, the engine's backpressure scheme throttles upstream computations in the DAG to avoid possible buffer overflows at the stream processor. However, backpressure schemes reduce the overall throughput of a stream processing application and should therefore be deferred or avoided

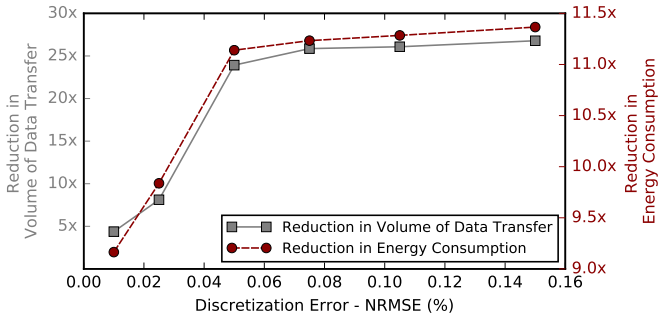


Fig. 13: Trade-off analysis of discretization error, volume of data transfer, and energy consumption at the edges.

if possible. Lower memory footprint of sketched streams used in Pebbles helps the buffers to accommodate more unprocessed messages; this postpones backpressure maneuver as shown in the results. This may provide ample time for the processor to resume its normal operations without triggering backpressure. In our benchmark, **Pebbles can hold ∼18× more messages compared to the other two processing modes** (we do not see a significant difference between one-message-at-a-time mode and the batched stream because batched streaming does not compact the unprocessed messages). Since Storm does not expose any APIs or metrics to capture backpressure triggering, we performed this benchmark only with Neptune.

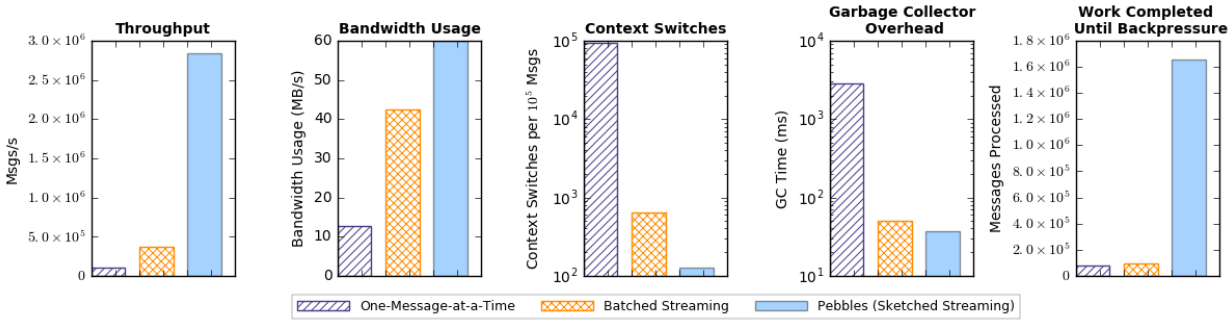
4.4.2 *Performance of In-sketch Operations*

We profiled the performance of in-sketch operations as shown in Figure 15. The throughput of different in-sketch operations were compared against the performance of the same operations performed on regular streams. For comparison purposes, we include the results corresponding to applying the regular operations on the materialized sketches as well. To isolate performance of the operations, the data was generated locally within the stream processor. We did not enable the parallel processing for sketch materialization and in-sketch operations. Except for round-robin partitioning, the in-sketch operations outperform regular streams significantly. Cloning of bit masks for individual bitmaps was inefficient, therefore the in-sketch implementation of round-robin partitioning did not perform well. We plan to improve the performance on this operation in future through efficient bit-shift operations.

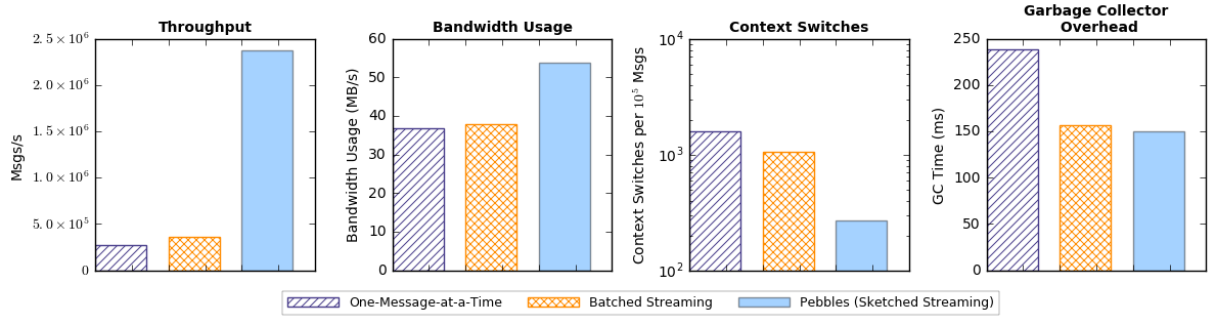
5 Related Work

Federated stream processing: Federated stream processing [16], [22], [14], [23] has been proposed as an alternative approach to the centralized stream processing in the presence of geo-distributed data sources. A subset of the operators of the stream processing graph are deployed at the edge devices close to the stream sources in order to locally process as many data. This approach is useful for providing low latency responses, location-aware services, and to reduce data transmissions to the center. In SpanEdge [16], a set of operators, called local operators, are deployed at the data centers closer to the edges and the remainder of the operators (called global operators) deployed at the center based on a classification provided by the user. Global operators running on data centers process derived partial data streams originating at the local operators such as aggregations. Renart et al. [22] proposes an edge based stream processing model to provide location-aware services where stream producers and consumers are connected via a content based subscription model. While federated stream processing fits well for certain use cases, it presents a few challenges including: ① limited processing capabilities at the edge devices that are inadequate to execute certain operators, ② dependencies on non-local data sources, and ③ an inability to reprocess past data. To circumvent these challenges, raw data streams (straight from the sources) or sometimes the derived data streams must be transferred to the cloud where our proposed approach based on sketched streams can be useful. It is also possible to incorporate the sketched streams within the federated stream processing solutions to efficiently transfer data from edge operators to operators in data centers.

Data processing at the edge: Apache Edgent [38], Amazon Greengrass [37], and Cisco Kinetic Edge and Fog Processing Module [64] are few examples of commercial and open source alternatives available for processing data at the edges of the



(a) With Neptune stream processing engine.



(b) With Apache Storm stream processing engine.

Fig. 14: Sketched streams improves the performance at the center compared to traditional *processing one-message-at-a-time* and *batched streaming* as shown with two stream processing engines: Neptune and Apache Storm.

network. These modules provide a programming model and an execution runtime for lightweight stream computations along with support for two-way communications with their counterparts running in the cloud. The connectivity with the cloud enables offloading processing, transferring derived data, and device management. These edge modules are often a component in a larger IoT echo system offered by the vendor and expected to reduce the data transfer to cloud and enable low latency processing. Apache Edgent and Amazon's Greengrass provide a functional programming style API to connect with data sources and manipulate the data streams. Despite several functional similarities with edge operators in federated stream processing systems, these edge modules are general purpose components that can be used to implement diverse use cases such as ETL pipelines. We posit that the our proposed methodology around sketched streams complements these edge modules - Sketched streams generated using Pebbles sketch algorithm can be used to further improve the data transfer costs between these edge modules and the center when applicable.

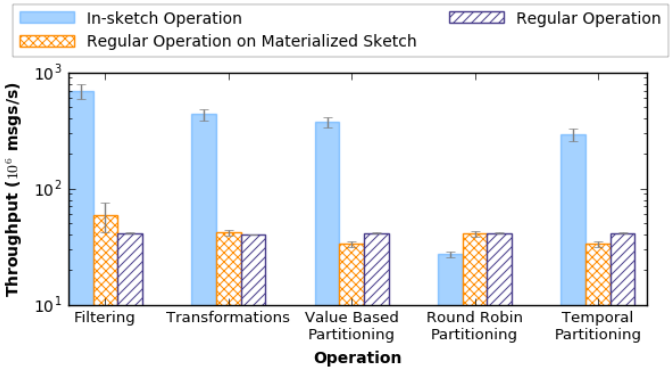


Fig. 15: Performance comparison in-sketch operations.

Data reduction at the edges: These techniques leverage seasonal patterns, predictable trends in data, and minor variations between adjacent observations. *Edge mining* [24], [25], [26] algorithms are designed to locally estimate a contextual state out of the streams and transmit only the changes to the aforementioned state to the receiver. For instance, the time-discounted histogram encoding algorithm [26] maintains a histogram representing the proportion of time spent on a set of states (e.g. walking and sitting for a stream produced by a gyroscope attached to a human) by locally processing a time series data stream. *Selective forwarding* techniques [27], [28], [65] attempt to reduce a stream into a fewer messages while ensuring the reconstructability of the stream at the destination with a certain accuracy. Traub et al.[65] proposes a read-fusion scheme where read requests from multiple applications are fused together into a single sensor read (through user-defined sampling functions) which is then shared between the applications at the center. These approaches are designed for single-feature streams whereas Pebbles sketching algorithm can natively handle multi-feature streams. Deligiannakis et al. [66] propose data reduction technique that works with multi-feature streams. At each sensor, consecutive readings are batched for each feature. Each batch is then transformed with respect to a base signal using the correlation of the feature values with the base signal. Batches are partitioned until a matching section of the base signal is found with a high correlation. At the destination, the streams are reconstructed based on the regression parameters and the base signal. These approaches focus only on the reducing the data transfer overhead and whereas Pebbles provides a holistic approach that encompasses data transfer and processing. *Compression* is another class of data reduction techniques that attempts to reduce the network footprint of the data streams by exploiting the low entropy of the data. Most lossless compression algorithms developed to run on energy constrained devices are dictionary based due to their lower

power profiles [29], [30], [10]. LTC [31] is a lossy temporal compression algorithm which exploits the linear trends in data. Unlike Pebbles, these algorithms often focus on single-feature streams and do not improve processing at the center.

Data Sketching: Frequency based sketches are a family of streaming algorithms designed to summarize the observed frequency distribution of a dataset [36]. They usually provide constant time update and query performance and require only a single pass over the dataset. Frequency based sketches are compact by design and trade off accuracy by providing guaranteed error bounds on estimated frequencies. Augmented Sketch [33], Count-Min [34], Misra-Gries algorithm [35], and Counting-Quotient filters [67] are a few examples of frequency based sketches. Augmented Sketch improves the existing frequency-based sketching algorithms by reducing the estimation error of the most frequent items using an adaptive pre-filtering scheme, especially in data streams with skewed distributions. Most frequent items are retained by a filter, which supports faster lookups through vector instructions, while remaining items are sketched using regular frequency based sketching algorithms. Counter based approaches [68], [69] are also used to summarize data streams where they are designed to capture the top-K elements in a stream. But these sketching algorithms cannot maintain ordering between events, and do not support general purpose processing — their applicability for ingestion in most stream processing use cases are limited.

6 Conclusions and Future Work

In this study, we presented a methodology and a reference implementation based on data sketching for data stream ingestion and processing in continuous sensing environments.

RQ-1: Sketched streams effectively reduce the data volumes at the edges of the network, resulting in lower data transfer costs and energy consumption. Our sketch-aware stream processing API significantly improve the performance of the stream processing applications while reducing resource footprints.

RQ-2: Pebbles sketches produce space-efficient representations of both single-feature and multi-feature streams through controlled trading off of the resolution of individual feature values. Uniquely, our algorithm can preserve the ordering between observations, handle anomalies effectively, and provide accuracy guarantees. Furthermore, Pebbles supports a richer set of operations useful in stream processing computations compared to existing frequency-based sketching algorithms.

RQ-3: Using sketches as the unit of data transfer/processing significantly improves throughput, bandwidth utilization, and tolerance to short-term performance hotspots, while reducing execution overhead (garbage collection, context switches). In-sketch operations provide memory-efficient, high throughput stream processing operations.

RQ-4: Space-efficient representation of streams allows storage of past data for extended periods of time. This enables reprocessing of past data to support new and updated applications. Pebbles is a drop-in extension to existing stream processing APIs enabling; (1) Pebbles applications to coexist with the regular stream processing applications, and (2) developing hybrid stream processing applications that leverage both APIs.

As part of future work, we will explore using proactive bin configuration generation schemes, deep integration with micro-batching systems such as Spark Streams [59], and further explore the applicability of Sketched streams in edge and federated stream processing usecases.

Acknowledgments: This research was supported by the National Science Foundation [OAC-1931363, ACI-1553685], the National Institute of Food and Agriculture [COL0-FACT-2019], and a Cochran Family Professorship.

References

[1] T. Stack. (2018) Internet of things (iot) data continues to explode exponentially. who is using that data and how? [Online]. Available: https://blogs.cisco.com/datacenter/internet-of-things-iot-data-continues-to-explode-exponentially-who-is-using-that-data-and-how

[2] National Oceanic and Atmospheric Administration. (2018) The North American Mesoscale Forecast System. [Online]. Available: http://www.emc.ncep.noaa.gov/index.php?branch=NAM

[3] Massachusetts Department of Transportation. (2017) MassDOT developers' data sources. [Online]. Available: https://www.mass.gov/massdot-developers-data-sources

[4] US Environmental Protection Agency. (2017) Air Data: Air Quality Data Collected at Outdoor Monitors Across the US. [Online]. Available: https://www.epa.gov/outdoor-air-quality-data

[5] M. Saeed *et al.*, "Multiparameter intelligent monitoring in intensive care ii (mimic-ii): a public-access intensive care unit database," *Critical care medicine*, vol. 39, no. 5, p. 952, 2011.

[6] S. R. Islam *et al.*, "The internet of things for health care: a comprehensive survey," *IEEE Access*, vol. 3, pp. 678–708, 2015.

[7] J. Fonollosa *et al.*, "Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring," *Sensors and Actuators B: Chemical*, vol. 215, pp. 618–629, 2015.

[8] C.-W. Tsai *et al.*, "Data mining for internet of things: A survey," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 77–97, 2014.

[9] J. Gubbi *et al.*, "Internet of things (iot): A vision, architectural elements, and future directions," *Future generation computer systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[10] F. Marcelloni *et al.*, "An efficient lossless compression algorithm for tiny nodes of monitoring wireless sensor networks," *The Computer Journal*, vol. 52, no. 8, pp. 969–987, 2009.

[11] C.-Y. Wan *et al.*, "Pump-slowly, fetch-quickly (psfq): a reliable transport protocol for sensor networks," *IEEE Journal on selected areas in Communications*, vol. 23, no. 4, pp. 862–872, 2005.

[12] Y. Sankarasubramaniam *et al.*, "Esrt: event-to-sink reliable transport in wireless sensor networks," in *Proceedings of the 4th ACM international symposium on Mobile ad hoc networking & computing*. ACM, 2003, pp. 177–188.

[13] S. Kamburugamuve *et al.*, "A framework for real time processing of sensor data in the cloud," *Journal of Sensors*, vol. 2015, 2015.

[14] P. Michalák *et al.*, "Path2iot: A holistic, distributed stream processing system," in *2017 International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2017, pp. 25–32.

[15] P. Edara *et al.*, "Asynchronous in-network prediction: Efficient aggregation in sensor networks," *ACM Transactions on Sensor Networks (TOSN)*, vol. 4, no. 4, p. 25, 2008.

[16] H. P. Sajjad *et al.*, "Spanedge: Towards unifying stream processing over central and near-the-edge data centers," in *Edge Computing, IEEE/ACM Symposium on*, 2016, pp. 168–178.

[17] M. Kleppmann, *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. "O'Reilly", 2017.

[18] J. Kreps. (2014-07-02) Questioning the lambda architecture. [Online]. Available: https://www.oreilly.com/ideas/questioning-the-lambda-architecture

[19] M. Satyanarayanan *et al.*, "The case for vm-based cloudlets in mobile computing," *IEEE pervasive Computing*, no. 4, pp. 14–23, 2009.

[20] B. Theeten *et al.*, "Chive: Bandwidth optimized continuous querying in distributed clouds," *IEEE Transactions on cloud computing*, vol. 3, no. 2, pp. 219–232, 2015.

[21] F. Bonomi *et al.*, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 2012, pp. 13–16.

[22] E. G. Renart *et al.*, "Data-driven stream processing at the edge," in *Fog and Edge Computing (ICFEC), 2017 IEEE 1st International Conference on*. IEEE, 2017, pp. 31–40.

[23] S. Esteves *et al.*, "Empowering stream processing through edge clouds," *SIGMOD Rec.*, vol. 46, no. 3, pp. 23–28, Oct. 2017.

[24] D. Goldsmith *et al.*, "The spanish inquisition protocolmodel based transmission reduction for wireless sensor networks," in *SENSORS, 2010 IEEE*. IEEE, 2010, pp. 2043–2048.

[25] J. Brusey *et al.*, "Postural activity monitoring for increasing safety in bomb disposal missions," *Measurement Science and Technology*, vol. 20, no. 7, p. 075204, 2009.

[26] E. I. Gaura *et al.*, "Bare necessitiesknowledge-driven wsn design," in *SENSORS, 2011 IEEE*. IEEE, 2011, pp. 66–70.

[27] D. Trihinas *et al.*, "Adam: An adaptive monitoring framework for sampling and filtering on iot devices," in *Big Data*. IEEE, 2015, pp. 717–726.

[28] W. Sherchan *et al.*, "Using on-the-move mining for mobile crowdsensing," in *Mobile Data Management (MDM), 2012 IEEE 13th International Conference on*. IEEE, 2012, pp. 115–124.

[29] C. M. Sadler *et al.*, "Data compression algorithms for energy-constrained devices in delay tolerant networks," in *Proceedings of the 4th international conference on Embedded networked sensor systems*. ACM, 2006, pp. 265–278.

[30] M. Oberhumer. minilzo: mini version of the lzo real-time data compression library. [Online]. Available: http://www.oberhumer.com/opensource/lzo/

[31] T. Schoellhammer *et al.*, "Lightweight temporal compression of microclimate datasets," 2004.

[32] A. Papageorgiou *et al.*, "Reconstructability-aware filtering and forwarding of time series data in internet-of-things architectures," in *International Congress on Big Data*. IEEE, 2015, pp. 576–583.

[33] P. Roy *et al.*, "Augmented sketch: Faster and more accurate stream processing," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1449–1463.

[34] G. Cormode *et al.*, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[35] J. Misra *et al.*, "Finding repeated elements," *Science of computer programming*, vol. 2, no. 2, pp. 143–152, 1982.

[36] G. Cormode, "Sketch techniques for approximate query processing," *Foundations and Trends in Databases. NOW publishers*, 2011.

[37] (2019) Aws iot greengrass: Bring local compute, messaging, data caching, sync, and ml inference capabilities to edge devices. [Online]. Available: https://aws.amazon.com/greengrass/

[38] (2016) Apache edgent: A community for accelerating analytics at the edge. [Online]. Available: http://edgent.apache.org/

[39] D. Locke, "Mq telemetry transport (mqtt) v3.1 protocol specification," *IBM developerWorks Technical Library*, 2010.

[40] G. C. Fox *et al.*, "Architecture and measured characteristics of a cloud based internet of things," in *2012 international conference on Collaboration Technologies and Systems (CTS)*. IEEE, 2012, pp. 6–12.

[41] Apache kafka: A distributed streaming platform. [Online]. Available: https://kafka.apache.org/

[42] Apache storm. [Online]. Available: http://storm.apache.org/

[43] T. Buddhika *et al.*, "Neptune: Real time stream processing for internet of things and sensing environments," in *Intl. Parallel and Distributed Processing Symposium*. IEEE, 2016, pp. 1143–1152.

[44] E. Parzen, "On estimation of a probability density function and mode," *The annals of mathematical statistics*, vol. 33, no. 3, pp. 1065–1076, 1962.

[45] M. Rosenblatt *et al.*, "Remarks on some nonparametric estimates of a density function," *The Annals of Mathematical Statistics*, vol. 27, no. 3, pp. 832–837, 1956.

[46] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software (TOMS)*, vol. 11, no. 1, pp. 37–57, 1985.

[47] C. C. Aggarwal, "On biased reservoir sampling in the presence of stream evolution," in *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment, 2006, pp. 607–618.

[48] J. Arfa. (2016) Virbs and sampling events from streams. [Online]. Available: http://tech.magnetic.com/2016/04/virbs-sampling-events-from-streams.html

[49] T. Akidau *et al.*, "Millwheel: fault-tolerant stream processing at internet scale," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[50] K. Wu, "Notes on design and implementation of compressed bit vectors," 2001.

[51] D. Lemire *et al.*, "Sorting improves word-aligned bitmap indexes," *Data & Knowledge Engineering*, vol. 69, no. 1, pp. 3–28, 2010.

[52] A. Colantonio *et al.*, "Concise: Compressed'n'composable integer set," *arXiv:1004.0403*, 2010.

[53] S. Chambi *et al.*, "Better bitmap performance with roaring bitmaps," *Software: practice and experience*, vol. 46, no. 5, pp. 709–719, 2016.

[54] J. Wang *et al.*, "An experimental study of bitmap compression vs. inverted list compression," in *Proceedings of the 2017 ACM SIGMOD*. ACM, 2017, pp. 993–1008.

[55] D. Cutting *et al.*, "Optimization for dynamic inverted index maintenance," in *Proceedings of the 13th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 1989, pp. 405–411.

[56] M. Zukowski *et al.*, "Super-scalar ram-cpu cache compression," in *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*. IEEE, 2006, pp. 59–59.

[57] V. N. Anh *et al.*, "Inverted index compression using word-aligned binary codes," *Information Retrieval*, vol. 8, no. 1, pp. 151–166, 2005.

[58] DEBS. (2014) ACM DEBS 2014 Grand Challenge: Smart homes. [Online]. Available: http://debs.org/debs-2014-smart-homes/

[59] M. Zaharia *et al.*, "Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters." *HotCloud*, vol. 12, pp. 10–10, 2012.

[60] M. Hirzel *et al.*, "A catalog of stream processing optimizations," *ACM Computing Surveys*, vol. 46, no. 4, p. 46, 2014.

[61] J. C. Process. (2012) Jsr 335: Lambda expressions for the javatm programming language. [Online]. Available: https://jcp.org/en/jsr/detail?id=335

[62] Y. Collet *et al.* (2013) Lz4: Extremely fast compression algorithm. [Online]. Available: https://lz4.github.io/lz4/

[63] J. Dean *et al.*, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[64] (2019) Cisco kinetic edge & fog processing module. [Online]. Available: https://www.cisco.com/c/dam/en/us/solutions/collateral/internet-of-things/kinetic-datasheet-efm.pdf

[65] J. Traub *et al.*, "Optimized on-demand data streaming from sensor nodes," in *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 2017, pp. 586–597.

[66] A. Deligiannakis *et al.*, "Compressing historical information in sensor networks," in *Proceedings of the 2004 ACM SIGMOD*, ser. SIGMOD '04. ACM, 2004, pp. 527–538.

[67] P. Pandey *et al.*, "A general-purpose counting filter: Making every bit count," in *SIGMOD*. ACM, 2017, pp. 775–787.

[68] G. Cormode *et al.*, "Finding frequent items in data streams," *Proc. of the VLDB Endowment*, vol. 1, no. 2, pp. 1530–1541, 2008.

[69] A. Metwally *et al.*, "Efficient computation of frequent and top-k elements in data streams," in *International conference on database theory*. Springer, 2005, pp. 398–412.

Thilina Buddhika received his Ph.D. from the Computer Science department at Colorado State University. His research interests are in the area of real time, high throughput stream processing specifically targeted to environments such as Internet of Things (IoT) and health care applications. Email: thilinab@cs.colostate.edu

Sangmi Lee Pallickara is an Associate Professor in the Department of Computer Science and a Cochran Family Professor at Colorado State University. She received her Masters and Ph.D. degrees in Computer Science from Syracuse University and Florida State University, respectively. Her research interests are in the area of large-scale scientific data management. She is a recipient of the NSF CAREER award. Email: sangmi@cs.colostate.edu

Shrideep Pallickara is a Professor in the Department of Computer Science at Colorado State University. His research interests are in the area of large-scale distributed systems. He received his Masters and Ph.D. degrees from Syracus University. He is a recipient of an NSF CAREER award. Email: shrideep@cs.colostate.edu