Living on the Edge: Data Transmission, Storage, and Analytics in Continuous Sensing Environments

THILINA BUDDHIKA, MATTHEW MALENSEK, SHRIDEEP PALLICKARA, and SANGMI LEE PALLICKARA, Colorado State University, USA

Voluminous time-series data streams produced in continuous sensing environments (CSEs) impose challenges pertaining to ingestion, storage, and analytics. In this study, we present a holistic approach based on data sketching to address these issues. We propose a hyper-sketching algorithm which combines discretization and frequency-based sketching to produce compact representations of the multi-feature, time-series data streams. We generate an ensemble of data sketches to make effective use of capabilities at the resource-constrained edge devices, the links over which data are transmitted, and the server pool where this data must be stored. The data sketches can be queried to construct datasets that are amenable to processing using popular analytical engines. We include several performance benchmarks using real-world data from different domains to profile the suitability of our design decisions. The proposed methodology can achieve up to $\sim 13\times$ and $\sim 2207\times$ reduction in data transfer and energy consumption at edge devices. We observe up to a $\sim 50\%$ improvement in analytical job completion times in addition to the significant improvements in disk and network I/O.

Additional Key Words and Phrases: data sketches, streaming systems, temporal data, Internet-of-Things, edge computing

ACM Reference Format:
Thilina Buddhika, Matthew Malensek, Shrideep Pallickara, and Sangmi Lee Pallickara. 2021. Living on the Edge: Data Transmission, Storage, and Analytics in Continuous Sensing Environments. 1, 1 (February 2021), 30 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Falling costs, network enhancements, and advances in miniaturization combined with improvements in the rates and resolutions at which measurements can be made have contributed to a proliferation of *continuous sensing environments* (CSEs). These CSEs manifest themselves in many forms such as Internet of Things (IoT), smart dust, fog computing, and observational devices (remote or in situ) used to monitor environmental [70], atmospheric [44], traffic [40], and other phenomena. In CSEs multiple features of interest are being monitored with observations including metadata (e.g. timestamps), the feature being measured (e.g. humidity, temperature, heart rate), and the *entity* being observed (e.g. person, geographical scope, topological information). These data and how they evolve over time contain a wealth of information that can be used to extract knowledge.

Voluminous data generated in CSEs can be attributed to two main factors: high data generation rates at individual entities and the vast number of entities. Data streams in these settings are typically time-series data [69] usually transferred from their sources to a centralized location for

Authors' address: Thilina Buddhika, thilinab@cs.colostate.edu; Matthew Malensek, malensek@cs.colostate.edu; Shrideep Pallickara, shrideep@cs.colostate.edu; Sangmi Lee Pallickara, sangmi@cs.colostate.edu, Colorado State University, Department of Computer Science, P.O. Box 1212, Fort Collins, Colorado, USA, 80523.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2021 Association for Computing Machinery.
XXXX-XXXX/2021/2-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

processing [56, 57] (typically a public cloud or a private cluster; for the remainder of the paper, we use the term cloud to refer to both private clusters and public clouds). Transferred data may get processed in near real-time using stream processing systems or as batches using batch processing systems. In certain cases, organizations arrange their data pipelines to facilitate both types of processing for a single data stream [34]. In case of analytic tasks modeled as batch processing tasks, the data may need to be stored for extended periods of time — e.g. when performing long term trend analysis. In this study, we focus on data transmission, storage, and subsequent analytics performed as batch processing jobs over time-series data streams generated in CSEs.

1.1 Challenges

Performing analytics on voluminous data streams generated at the edges of the network introduces challenges in the *ingestion*, *storage*, and *analytics* phases leading up to it.

- *Energy consumption at the edge devices:* Communication is the dominant energy consuming factor for sensing devices [22, 41] requiring frugal transmissions.
- *Network bandwidth:* Edge devices are usually connected to the cloud via wide area networks with limited bandwidth [59]. Also in the case of public clouds, customers are billed for the amount of data transferred into the cloud from external sources. Continuous, high-velocity data streams incur network congestions and increased bandwidth and data transfer costs.
- *Storage provisioning:* The cumulative data generation rate in a CSE with multitude of sensors may outpace the rate at which the data can be written to the disks in the available cloud servers. Also continually increasing the capacity of the storage cluster to match the ever increasing storage demand of streaming datasets is challenging, and in several cases not economically viable.
- *Accessing stored data with analytical engines:* Stored data should be readily available for analytics using various analytical engines such as Apache Spark [3] and Apache Hadoop [4]. The storage system should support efficient retrieval of data while accounting for the speed differential of the memory hierarchy with disk I/O being several orders of magnitude slower than memory [13, 48].

Several attempts have been made to address these individual challenges in isolation. Availability of limited processing and storage capacities at the edges of the network through sensor network aggregator nodes [39], cloudlets [60], and distributed telco clouds [66] has enabled preprocessing of data streams closer to the source before transferring them to the cloud. This work can be broadly categorized as ① data reduction techniques (edge mining [17, 22, 27, 29, 63, 72], sampling [67, 68], compression [39, 49, 58, 61]) and ② federated processing [32, 38]. Data reduction techniques at the edges leverage recurring patterns, the gradually evolving nature of data streams, and low entropy of feature values to reduce the data volumes that are transferred. Federated processing techniques deploy a portion of the data processing job in close proximity to the data sources to reduce data transfers to the remainder of the job — for instance filtering and aggregation executed on edge devices can replace the raw data streams with derived streams with a smaller network footprint. On the storage front, time-series databases [7, 9–11] are specifically designed for time-series data streams while organizations sometimes repurpose distributed file systems, relational databases, and NoSQL data stores to store time-series data streams [45].

We identify the following limitations in existing work.

- *Limited focus on holistic solutions encompassing ingestion, storage, and analytics:* Current solutions mostly focus only on addressing a single aspect of the problem — for instance, upon ingestion edge reduction techniques often reconstruct the original data stream at the cloud [51] therefore not addressing the storage provisioning issues.

- *Limited applicability:* Edge devices often have limited computation power and ephemeral/semi-persistent storage [15] hence the types of processing tasks feasible at the edge devices are limited.
- *Designed for single-feature streams:* Most data reduction techniques are designed for single-feature streams, but usually multiple phenomena are monitored simultaneously in modern CSEs.
- *Focus entirely on current application needs:* Preprocessing data at the edges should not preclude the use of ingested data in future applications. For instance, edge mining techniques only forward a derived data stream tailored for current application requirements which may leave out portions of the data space critical for future application needs.
- *Limited aging schemes:* Most time-series databases do not offer a graceful aging scheme to reclaim storage space as the size of the dataset grows. Common practices are deletion, reducing the replication level, and using erasure coding for cold data [13, 45]. These schemes affect the data reliability and the retrieval times. Some time series databases support aging by replacing cold data with aggregated values [9] — while this is effective in controlling the growth of the data it also reduces the usability of aged data.
- *Poor integration between time-series databases and analytical engines:* Query models of the time-series databases are designed to answer specific user queries. Extracting a portion of the dataset to run complex analytic jobs such as learning jobs are not natively supported.

1.2 Research Questions

Research questions that guide this study include

RQ-1: *How can we develop a holistic methodology to address challenges pertaining to ingestion, storage, and analysis of time-series data streams?* Individual data items may be multidimensional, encapsulating observations that comprise multiple features of interest.

RQ-2: *How can we support reducing the network and storage footprint of time-series data streams without enforcing restrictions on future application requirements?*

RQ-3: *How can we cope with the increasing storage capacity demands of time-series data streams by effectively leveraging the storage hierarchy and aging cold data?*

RQ-4: *How can we support exploratory analytics by efficiently identifying and retrieving portions of the feature space and interoperating with analytical engines?*

1.3 Approach Summary

Our framework, called *Gossamer*, enables analytics in CSEs by ensuring representativeness of the data and feature space, reducing network bandwidth and disk storage requirements, and minimizing disk I/O. We propose a hyper-sketching algorithm called *Spinneret*, combining discretization and frequency based sketching algorithms, to generate space-efficient representations of multi-feature data streams in CSEs. Spinneret is the primary data unit used for ingestion and storage within Gossamer. In Gossamer, we leverage fog computing principles — Spinneret sketches are generated at the edges of the network and an ensemble of Spinneret instances are stored in a server pool maintained in the cloud. Spinneret sketches are generated *per segment*, *per entity*. We define a segment as the configured smallest unit of time for which a Spinneret instance is constructed for a particular stream. Multiple Spinneret sketches corresponding to smaller temporal scopes can be aggregated into a single instance to represent arbitrary temporal scopes.

Spinneret performs a controlled reduction of resolution of the observed feature values through discretization. Features are discretized via a binning strategy based on the observed (and often known) probability density functions in the distribution of values. This is true for several natural (temperature, humidity), physiological (body temperature, blood oxygen saturation), commercial (inventory, stock prices), and experimental phenomena. The discretized feature vector representing a set of measurements is then presented for inclusion into the relevant Spinneret instance. Spinneret uses a frequency based sketching algorithm to record the observed frequencies of the discretized feature vectors. Spinneret stores necessary metadata to support querying the observed discretized feature vectors for each segment.

Ancillary data structures at each storage node in the cloud extract and organize metadata from Spinneret sketches as they are being ingested. These metadata are organized such that they capture the feature space and are amenable to query evaluations. The frequency data (sketch payload) embedded within Spinneret sketches are organized within server pools following a temporal hierarchy to facilitate efficient retrieval and aging. Our aging scheme is designed by leveraging sketch aggregation — several, continuous Spinneret sketches can be aggregated into a single Spinneret sketches to reclaim space by trading off the temporal resolution and estimation accuracy. The result of a query specified over the managed data space is a virtual dataset (called a *Scaffold*) that organizes metadata about segment sketches that satisfy the specified constraints.

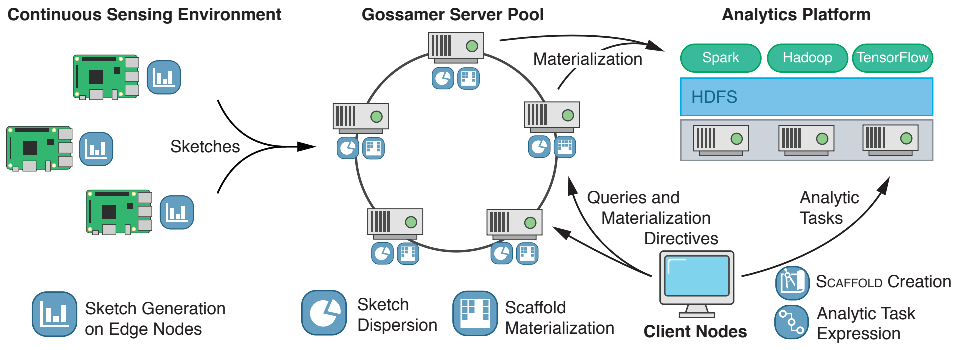
The Scaffold abstraction is key to enabling analytics by hiding the complexities of distributed coordination, memory residency, and processing. Materialization of a Scaffold results in the generation of an *exploratory dataset*. The same Scaffold may be materialized in different ways to produce diverse exploratory datasets. Materialization of a Scaffold involves generation of synthetic datasets, identification of shards, and aligning distribution of shards with the expected processing. Shards represent indivisible data chunks that are processed by tasks comprising the analytics job. We materialize shards in HDFS [8], which provides a strong integration with analytical engines such as Hadoop and Spark.

1.4 Paper Contributions

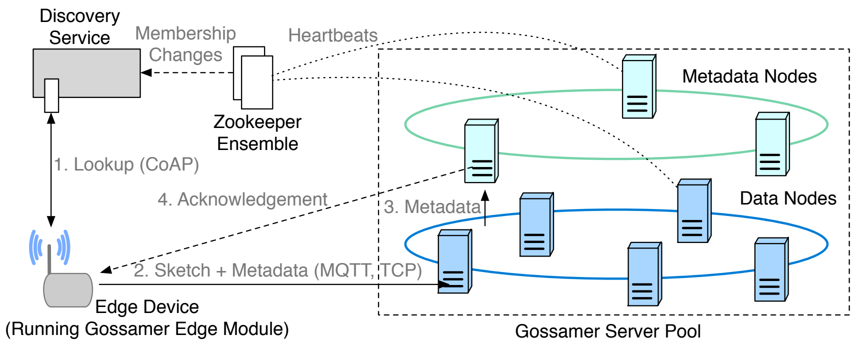
Our methodology substantially alleviates data storage, transmission, and memory-residency. Comprehensively reducing resource footprints reduces contention for disk, network links, and memory. More specifically, our methodology:

- Presents a holistic approach based on data sketching to address ingestion, storage, and analytic related challenges without constraining future application requirements
- Introduces Spinneret — a novel hyper-sketching algorithm providing a space-efficient representation of multi-feature time series streams to reduce the data transfers and storage footprints.
- Reduces the data transfers and energy consumption at the edges of the network through sketch based preprocessing of streams while interoperating with dominant edge processing frameworks such as Amazon IoT and Apache Edgent.
- Proposes an efficient aging scheme for time-series streaming datasets to provide memory residency for relevant data while controlling the growth of the stored dataset
- Improves the exploratory analysis through efficient retrieval of relevant portions of the data space, sharded synthetic dataset generation, and integration with analytic engines.

We evaluated our approach using multiple datasets from various domains including industrial monitoring, smart homes, and atmospheric monitoring. Based on our benchmarks, Spinneret is able to achieve up to $\sim 2207\times$ and $\sim 13\times$ reduction in data transfer and energy consumption during ingestion. We observed up to $\sim 99\%$ in improvement in disk I/O, $\sim 86\%$ in improvement in network I/O, and $\sim 50\%$ in improvement in job completion times compared to running analytical jobs on data stored using existing storage schemes. We also performed a series of analytic tasks on synthetic datasets generated by Gossamer and compared against the results from the original datasets to demonstrate its applicability in real world use cases.



(a) High-level overview of Gossamer.



(b) System architecture.

Fig. 1. Gossamer relies on sketches as the primary construct for data transmission and storage.

1.5 Paper Organization

We present our methodology in Section 2. System benchmarks are presented in Section 3. In Section 4, we demonstrate suitability using real-world analytical tasks. Sections 5 and 6 discuss related work and conclusions respectively.

2 METHODOLOGY

The aforementioned challenges necessitate a holistic approach encompassing efficient data transfer from the edge devices, effective storage, fast retrievals, and better integration with analytical engines. To accomplish this, we

1. Generate sketches at the edges: We rely on an ensemble of Spinneret instances; a Spinneret instance is generated at regular time intervals at each edge device. To construct a Spinneret instance, multidimensional observations are discretized and their frequencies are recorded using frequency-based sketch algorithms. Spinneret instances (sketches and their metadata), not raw data, are transmitted from the edges. [**RQ-1**, **RQ-2**]

2. Effectively organize the server pool: Sketches and the metadata included within Spinneret instances need to be organized such that they are amenable to query evaluations and data space explorations. The server pool must ensure load balancing, aging of cold data, facilitate memory residency, and support low-latency query evaluations and fast retrieval of sketches. [**RQ-1**, **RQ-2**,

RQ-3]

3. Support construction of exploratory datasets that serve as input to analytical engines. A first step to creating exploratory datasets is the construction of Scaffolds using queries. A scaffold comprises data from several sketches. Exploratory datasets are created from scaffolds using materialization that encompasses generating synthetic data, creating shards aligned with expected processing, and supporting interoperation with analytical engines. [**RQ-1**, **RQ-4**].

Key architectural elements of Gossamer and their interactions are depicted in Figure 1.

Gossamer edge module is deployed on edge devices to convert an observational stream into a stream of Spinneret instances. A Gossamer edge module may be responsible for a set of proximate entities. Gossamer edge module expects an observation to include the CSE and entity identifiers, timestamp (as an epoch), and the series of observed feature values following a predetermined order. For instance, in a sensor network, an aggregator node may collect data from a set of sensors to construct an observation stream and relay it to a Gossamer edge module deployed nearby. Also Gossamer edge module can be deployed within various edge processing runtimes such as Amazon's Greengrass [6] and Apache Edgent [2]. We do not discuss the underlying details of this integration layer as it is outside the core scope of the paper.

Gossamer servers are used to store Spinneret sketches produced by the edge modules. The communication between Gossamer servers and edge modules take place either using MQTT [36] or TCP. MQTT is a lightweight messaging protocol designed for machine-to-machine (M2M) communications in constrained device environments, especially with limited network bandwidth.

Discovery service is used by edge modules to lookup the Gossamer server responsible for storing data for a given entity. The discovery service exposes a REST API to lookup Gossamer servers (for sketches and metadata) responsible for an entity through the Constrained Application Protocol (CoAP) [62]. CoAP is a web transfer protocol, similar to HTTP, designed for constrained networks.

2.0.1 Microbenchmarks, Setup, and Data. We validated several of our design decisions using microbenchmarks that are presented inline with the corresponding discussions. We used Raspberry Pi 3 model B single board computers (1.2 GHz, 1 GB RAM, 160 GB flash storage) as the edge devices running Arch Linux, F2FS file system, and Oracle JRE 1.8.0_65. The Gossamer server nodes were running on HP DL160 servers (Xeon E5620, 12 GB RAM).

For microbenchmarks, data from NOAA North American Mesoscale Forecast System (NAM) [44] for year 2014 was used to simulate a representative CSE where 60922 weather stations were considered as entities within the CSE. We considered 10 features including temperature, atmospheric pressure, humidity, and precipitation. This dataset contained 366,332,048 (frequency - 4 observations/day) observations accounting for a volume of ~221 GB.

2.1 Spinneret — A Sketch in Time (RQ-1, RQ-2)

We reduced data volumes close to the source to mitigate strains on the downstream components. Reductions must preserve representativeness of the data space, keep pace with arrival rates, and operate at edge devices. As part of this study, we have devised a hyper-sketching algorithm — *Spinneret*. It combines micro-batching, discretization, and frequency-based sketching algorithms to produce compact representations of multi-feature observational streams. Each edge device produces an ensemble of Spinneret sketches, one at configurable, periodic intervals (or *time segments*). At an edge device, an observational stream is split into a series of non-overlapping, contiguous, time segments creating a series of micro-batches. Observations within each micro-batch is discretized and the frequency distribution of the discretized observations are captured using a frequency based sketching algorithm. Producing an ensemble of sketches allows us to capture variations in the data space over time. Figure 2 illustrates a Spinneret instance.

2.1.1 Discretization. Discretization is the process of representing the feature values within an observation at lower resolutions. More specifically, discretization maps a vector of continuous values to a vector of bins. As individual observations are available to the Gossamer edge module, each (continuous) feature value within the observation is discretized and mapped to a bin. The bins are then combined into a vector, called as the *feature-bin combination.* Discretization still maintains how features vary with respect to each other.

Feature values in most natural phenomena do not change significantly between the consecutive measurements. This particular characteristic lays the foundation for most of the data reduction techniques employed at the edges of the network. There is a high probability that consecutive values for a particular feature are mapped to the same bin. This results in a lower number of unique feature-bin combinations within a time segment which reduces the data volume in two ways:

(1) Curtails the growth of metadata: Frequency data (sketch payload) within a Spinneret sketch instance maintains a mapping of observations to their frequencies, but not the set of unique observations. This requires maintaining metadata about the set of unique observations alongside the frequency data. Otherwise, querying a Spinneret instance requires an exhaustive search over the entire key space. Given that the observations are multidimensional, the set could grow rapidly because a slight change in a single feature value could result in a unique observation. To counteract such unimpeded growth, we compromise the resolution of individual features within an observation through discretization.
(2) Reduces the size of the sketch instance: Lower number of unique items require a smaller data container to provide a particular error bound [31].

For example, let's consider a simple stream with two features `A` and `B`. The bin configurations are `(9.9, 10.1, 10.3)` and `(0.69, 0.77, 0.80, 0.88)` for `A` and `B` respectively. The time segment is set to 2 time units. Let's consider the stream segment with the first three elements. Each element contains the timestamp followed by a vector of observed values for features A and B.

$$[0, \langle 10.01, 0.79 \rangle], [1, \langle 10.05, 0.78 \rangle], [2, \langle 9.89, 0.89 \rangle]$$

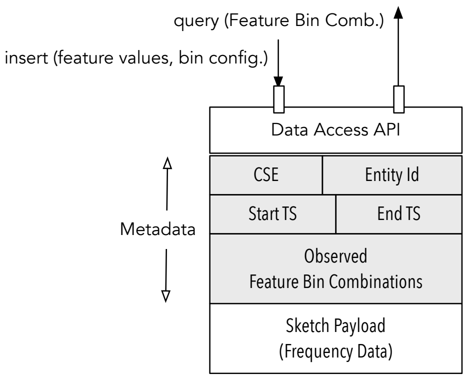

query (Feature Bin Comb.)
insert (feature values, bin config.)
Data Access API
CSE
Entity Id
Start TS
End TS
Metadata
Observed Feature Bin Combinations
Sketch Payload (Frequency Data)

Fig. 2. An instance of the Spinneret sketch. Spinneret is a hyper-sketching algorithm designed to represent observations within a stream segment in space-efficient manner by leveraging discretization and frequency based sketching algorithm.

Because we use a segment length of 2 time units, our algorithm will produce two microbatches for the intervals [0,2) and [2,4). There will be a separate Spinneret instance for each microbatch. Let's run our discretization algorithm on the first observation. The value for feature `A` (`10.01`) maps to the first bin `[9.9, 10.1)` in the corresponding bin configuration. Similarly second feature value `0.79` maps to the second bin `[0.77, 0.80)` of the feature `B`'s bin configuration. The identifiers of the two bins for features A and B are then concatenated together to generate the feature bin combination — i.e. `00` and `01` are combined together to form the feature bin combination `0001`. Similarly, the second observation in the stream is converted to the same feature bin combination `0001`. Then the sketch instance within the Spinneret instance for the first time segment is updated. The frequency for FBC `0001` is incremented by `2`. The feature bin combination `0001` is added to the metadata of the Spinneret instance.

For each feature, these bins should be available in advance at the edge device. The bins are either precomputed based on historical data, or may be specified by domain experts depending on the expected use cases. The bins are generated once for a given CSE and shared among all the participating edge devices. The requirements for a bin configuration are ① bins should not overlap and ② they should collectively cover the range of possible values for a particular feature (the range supported by the deployed sensor). When discretizing based on historical data, we have in-built support for binning based either on equal width or equal frequency. In the case of equal-width binning, the range of a feature value is divided by the number of required bins. With equal-frequency binning, we use kernel density estimation [52] to determine the bins. There is a trade-off involving the number of bins and the representational accuracy. As more bins are added, discretization approximates the actual non-discretized value range very closely, thus preserving the uniqueness of observations that differ ever so slightly. Number of bins is configured such that the discretization error is maintained below a given threshold. For instance, in our benchmarks we used normalized root mean square error (NRMSE) of 0.025 as the discretization error threshold.

2.1.2 Storing Frequency Data. We use frequency-based sketching algorithms to store the frequency data of the feature-bin combinations. Frequency-based sketching algorithms ① summarize the frequency distributions of observed values in a space-efficient manner ② trade off accuracy but provide guaranteed error bounds; ③ require only a single pass over the dataset, and ④ typically provide constant time update and query performance [19].

We require suitable frequency-based sketching algorithms to satisfy two properties in order to be considered for Spinneret.

(1) Lightweight - the computational and memory footprints of the algorithm should not preclude their use on resource constrained edge devices.
(2) Support for aggregation - the underlying data structure used by the algorithm to encode sketches should support aggregation allowing us to generate a sketch for a longer temporal scope by combining sketches from smaller scopes. Linear sketching algorithms satisfy this property [20].

Algorithms that satisfy this selection criteria include the Count-Min [20], frequent items sketch (Misra-Gries algorithm) [31, 43], and Counting-Quotient filters [50]. Spinneret leverages probabilistic data structures used in the aforementioned frequency based sketching algorithms to generate compact representations of the observations within segments with guaranteed bounds on estimation errors. Currently, we support Count-Min (Spinneret with probabilistic hashing) and the frequent items sketch (Spinneret with probabilistic tallying) and include support for plugging-in other sketching algorithms that meet the criteria.

Spinneret with probabilistic hashing: Count-min sketch uses a matrix of counters (m rows, n columns), and an m number of pair-wise independent hashing functions. Each of these hash functions uniformly maps the input domain (all possible feature-bin combinations within a time segment in case of Spinneret) into a range $\{0, 1, .., n-1\}$. During the ingestion phase, each of these hash functions (suppose hash function h_i corresponds to i^{th} row; $0 \leq i < m$) hashes a given key (feature-bin combination in the case of Spinneret), to a column j ($0 \leq j < n$); followed by an increment of the counter at cell (i, j). During lookup operations, the same set of hashing operations are applied on the key to identify the corresponding m cells, and the minimum of the m counters is picked as the estimated frequency to minimize possible overestimation errors due to hash collisions. It should be noted that the discretization step significantly reduce the size of the input domain, therefore reducing the probability of hash collisions. The estimation error of a Count-Min sketch can be controlled through the dimensions of the underlying matrix [19]. With a probability of $1 - \frac{1}{2^m}$, the upper bound for the estimation error is:

$$\frac{2N}{n} \quad \textit{[N: Sum of all frequencies]} \tag{1}$$

Spinneret with probabilistic tallying: Frequent items sketch internally uses a hash map, that is sized dynamically as more data is added [31]. The internal hash map has an associated load factor, l, (0.75 in the reference implementation we used), which determines the maximum number of feature-bin combinations and counter pairs (C) maintained at any given time based on its current size (M).

$$C = l \times M$$

When the entries count exceeds C, the frequent items sketch will decrements all counters by an approximated median and gets rid of the negative counters, therefore favoring the feature-bin combinations with higher frequencies. The estimation error of a frequency items sketch is defined in terms of an interval surrounding the true frequency. With x number of entries, the width (I) of this interval is:

$$I = \begin{cases} 0, & if\ x < C \\ 3.5 \times \frac{N}{M}, & Otherwise \quad \textit{[N: Sum of all frequencies]} \end{cases} \tag{2}$$

Similar to the case with Count-Min, over the use of discretization curbs the growth of unique entries in a Frequent Items sketch (such that $x < C$), therefore reducing the estimation error.

Once the time segment expires, current Spinneret instance is transferred to the Gossamer server pool for storage. A Spinneret instance is substantially more compact than the raw data received over the particular time segment. Data sketching reduce both the rate and volume of data that needs to be transferred by the edge devices. This reduction in communications is crucial at edge devices, where communications are the dominant energy consumption factor compared to local processing [22, 41]. It also reduces the bandwidth consumption (between the edges and the cloud) and data transfer and storage costs at the cloud.

For the remainder of this paper, we refer to the frequency payload embedded in a Spinneret instances as the sketch. Feature bin combinations, temporal boundaries, and entity information in a Spinneret instances will be collectively referred to as metadata.

2.1.3 Design choice implications. Discretization limits the applicabilty of our methodology only for streams with numeric feature values which we believe still covers a significant portion of use cases. By using Spinneret as the construct for data transfer and storage, we make the following controlled tradeoffs: ① reduced resolution of individual feature values due to discretization, ② estimated frequencies due to sketching, ③ ordering of observations *within* a time segment is not preserved, and ④ the finest temporal scope granularity within query predicates is limited to the length of the time segment.

Higher resolution can be maintained for discretized feature values by increasing the number of bins in at the expense of lower compaction ratios. The downside is the increase in the size of the input domain, which may lead to higher estimation errors. By adjusting the duration of the time segment, the impact of other trade-offs can be controlled. For instance, shorter time segments lower the estimation errors (through lowering N in equations 1 and 2) and support fine-grained temporal queries, but increase data storage and transfer costs. To maintain the estimation errors below the expected thresholds, users can configure the appropriate parameters of the underlying sketch based on the expected data rates (N). Further, the nature of the use cases is also factored in when selecting the sketching algorithm. For instance, the Misra-gries algorithm is preferable over Count-Min for use cases that focus on trend analysis use cases. Our methodology can be easily extended to maintain error thresolds under dynamic data rates (including bursts) by supporting dynamic time segment durations. A Spinneret instance will be considered complete if one of the following conditions are satisfied: ① the configured time segment duration is complete, or ② the number of maximum observations are complete. Under this scheme, in case of the bursts in data rates, the data for a time segment is represented by several sketch instances instead of a single sketch. Remainder of the ingestion pipeline does not need to change as the inline metadata of a sketch already carries the temporal boundaries.

2.1.4 Microbenchmark. We profiled the ability of the edge devices and sketches to keep pace with data generation rates. Our insertion rates include the costs for the discretization, sketch initializations and updates thereto. NOAA data from year 2014 with 10 features was used for this benchmark with a time segment length of 1 hour. The mean insertion rate during a time segment for the Spinneret with probabilistic hash was 43891.13 observations/s (std. dev.: 1261.76), while it was 60780.97 observations/s (std. dev.: 2157.43) for the Spinneret with probabilistic tally at the Raspberry Pi edge nodes.

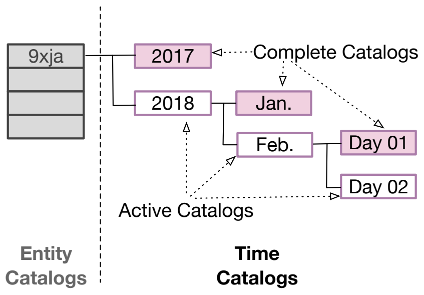
2.2 From the Edges to the Center: Transmissions (RQ-1, RQ-2)

Transmission of Spinneret instances from the edge devices to the Gossamer server pool target efficiency, minimizing redirection of traffic within the server pool, and coping with changes to the server pool. All edge device transmissions are performed using MQTT (by default) or TCP. Given that each Gossamer server is responsible for a set of entities, edge modules attempt to deliver the data to the correct server in order to reduce internal traffic within the server pool due to data redirections. The discovery service is used to locate the server node(s) responsible for holding the sketched data for a given entity. The discovery service tracks membership changes within the server pool using ZooKeeper [30] and deterministically maps entity identifiers to the appropriate server (based on hashing as explained in Section 2.3.4). ZooKeeper is a production-ready distributed coordination service widely used to implement various distributed protocols. In a Gossamer deployment, we use the ZooKeeper ensemble for two main use cases: ① node discovery within the Gossamer DHT and ② to update the discovery service on cluster changes. The discovery service relieves the edge modules from the overhead of listening for membership changes and decouples the edge layer from the Gossamer server pool. The mapping information is cached and reused by edge devices. If there is a message delivery failure (server crashes) or redirection (addition of new servers or rebalancing), then the cache is invalidated, and a new mapping is retrieved from the discovery service.

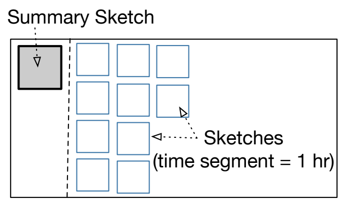
Data structures used to encode frequency data are amenable to compression, further reducing the data transfer footprints. For instance in the case of Spinneret with probabilistic hash, in most time segments a majority of the cells maintained by a count-min sketch are zeros, making them sparse matrices. For NOAA data [44](introduced in Section 2.0.1) for year 2014 with 60922 entities using 1 day as the time segment length, 83.7% of the matrices were found to have at least 7977 empty cells (out of 10000 cells). This is mainly due to duplicate feature-bin combinations that result from less variability in successive feature values (in most natural phenomena) that is amplified by our discretization. This sparsity benefits from both binary compression schemes and compact data structures, such as the compressed sparse raw matrix format for matrices. Based on our microbenchmarks at the edge devices, binary compression (GZip with a compression level of 5) provided a higher compression ratio (23:1) compared to compressed sparse raw format (4:1). However, the compressed sparse raw matrix format aligns well with our aging scheme where multiple sketches can be merged without decompression, making it our default choice.

2.2.1 Implementation Limitations. Gossamer edge module API supports movement of entities by decoupling the entities from the edge module. The current implementation of the edge module can be used to support cases where the edge module is directly executed on the entity (e.g. a mobile application). However, it can be extended to support the situations where entities temporarily connect with an edge module in close proximity for ingesting data to the center. Supporting this feature requires some improvements such as transferring incomplete segments corresponding to the disengaged entities, and merging partial Spinneret instances at the storage layer.

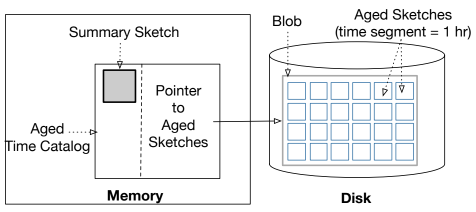
In our current implementation, we do not address crash failures of edge modules. However, communication failures are handled through repeated data transfer attempts (e.g. higher QoS levels of MQTT), deduplication at the server side, and support for out-of-order data arrivals.



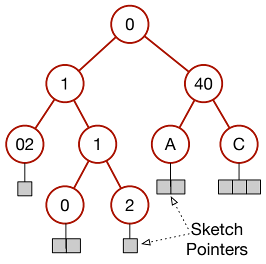
(a) Sketches for an entity are stored under an entity catalog. Within an entity catalog, there is a hierarchy of time catalogs.



(b) A time catalog stores sketches for a particular temporal scope and a summary sketch that aggregates them.



(c) Aging moves individual sketches within a time catalog to the disk and retains only the summary sketch in memory.



(d) Metadata tree is an inverted index of observed feature-bin combinations organized as a radix tree.

Fig. 3. Organization of Spinneret instances within a Gossamer node.

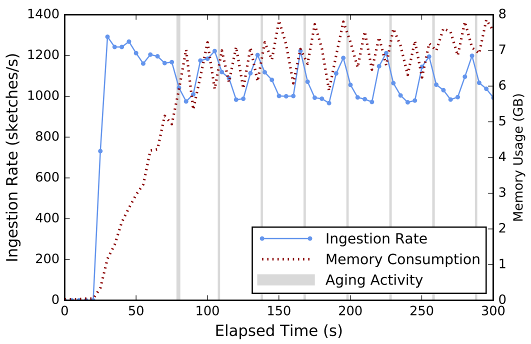


Fig. 4. Ingestion rate vs. memory usage at a data node. Sustaining high ingestion rates requires efficient aging.

2.3 Ingestion - Storing Data at the Center (RQ-1, RQ-3)

Sketches and metadata included in Spinneret instances are stored in the Gossamer server pool. We describe how we (1) store sketches, (2) collate metadata, and (3) organize the server pool to support fast query evaluations and data retrievals. Sketches or metadata from a single entity are stored deterministically at a particular node while a server holds data from multiple entities.

2.3.1 Storing Sketches. Sketches are organized in a two-tier *catalog* structure within a sketch storage server as shown in Figure 3a. Catalogs are instrumental for the functioning of our aging scheme. Sketches corresponding to an entity are stored within a dedicated *entity catalog*. Within each entity catalog, a hierarchy of *time catalogs* are maintained encompassing different temporal scopes. Time catalogs at the same level of the hierarchy are non-overlapping, and the union of finer-grained time catalogs (child catalogs) forms an upper-level time catalog (parent catalog). The finest-granular time catalog is one level higher than the entity segment duration. For example, in Figure 3a, the finest time catalog has a scope of 1 *day* and acts as a container for sketches generated for the time segments of 1 *hour*. The next level of time catalogs corresponds to *months* and holds *daily* time catalogs. Users can define the time catalog hierarchy for a CSE and may not necessarily follow the natural temporal hierarchy.

The finest-grained time catalog is considered *complete* when it has received sketches corresponding to all time segments that fall under its temporal scope. For example, in Figure 3a, time catalog

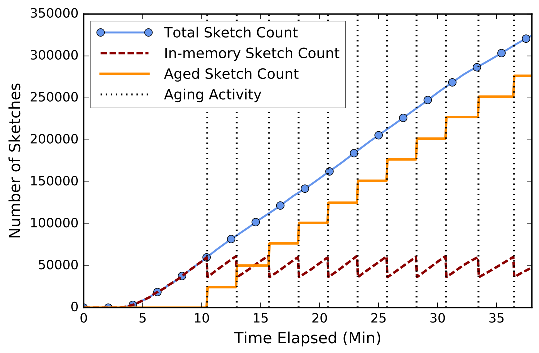


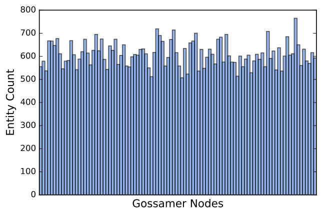
Fig. 5. Number of sketches maintained at a node over time. In-memory sketch count remains approximately constant whereas the aged sketches count increase.

for a day is considered complete when it has received 24 hourly sketches. A higher-level time catalog is complete when all its child time catalogs are complete. Every higher-level time catalog maintains a summary sketch of the currently completed child catalogs that is updated when a child time catalog is completed. Similarly, the finest-grained catalog also maintains a summary sketch calculated over all the received sketches as shown in Figure 3b. Summary sketch is the aggregation of summary sketches of its child catalogs (if it's calculated at a higher-level catalog) or the individual sketches if it is at the finest grained catalog. A summary sketch is updated in an online manner by merging the current summary sketch with the newly arrived sketch or the summary of the completed child catalog without bulk processing the individual sketches.

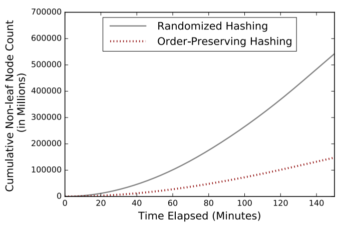
2.3.2 Aging. Aging in Gossamer is responsible for (1) Ensuring memory residency for most relevant data and (2) Reclaiming disk space. In both situations sketches of fine-grained temporal scopes are replaced by a summary sketch corresponding to the aggregated temporal scope. We use catalogs to implement our hierarchical aging scheme; fine-grained sketches in a catalog are replaced by its summary sketch.

All entity catalogs are memory resident. Upon creation, a time catalog is considered *active* and placed in memory. Over time, as more sketches are ingested the catalog hierarchy expands; this necessitates maneuvers to keep the memory consumed by the time catalogs below the thresholds. We use aging to reclaim memory by migrating complete time catalogs to disk. The Gossamer aging scheme prunes the in-memory time catalog hierarchy starting from the finest-grained time catalogs. Aging a complete, finest-grained time catalog involves migrating the individual sketches to disk and keeping only the summary sketch in memory. A higher-order, complete time catalog becomes eligible for aging only when all its child time catalogs are aged. Aging a higher-order time catalog involves moving the summary sketches of the child time catalogs to disk and keeping the summary sketch in memory. The total memory available for in-memory sketches is proportional to their depth in the time catalog hierarchy where most memory is allocated for finest-grained time catalogs. A reactive threshold-based scheme is used to trigger the aging process based on the allocated memory utilization levels (by default, we target 66% utilization). Selection of time catalogs for aging is done based on the criteria provided by the user for a given CSE. By default, Gossamer ages older time catalogs to disk first, leaving most recent time catalogs in memory. Users can override the default with custom directives; e.g. prioritizing certain entities over others. Catalogs from the most coarse-grained level are completely migrated to the disk (without maintaining a summary sketch) using the same criteria when it exceeds the alloted memory thresholds. For every sketch migrated to disk, the catalog maintains pointers so that it can retrieve the migrated sketch from disk if required. This is depicted in Figure 3c. This design enables accessing a more coarse-grained in-memory summary sketch with low latency or accessing finer-grained individual sketches with a higher latency depending on the use case.

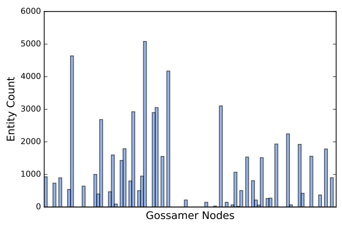
Aging should be efficient to keep pace with fast ingestion rates. Given that aging involves disk access and the recent developments in datacenter network speeds compared to disk access speeds [13], effective aging during high ingestion rates presents unique challenges. Instead of writing individual sketches as separate files, we perform a batched write by grouping multiple sketches together into a larger file (blobs), which reduces the disk seek times [48]. This approach simplifies maintaining pointers to individual sketches in an aged-out catalog. Instead of maintaining a set of file locations, only the file location of the blob and a set of offsets need to be maintained. We use multiple disks available on a machine to perform concurrent disk writes. Faster disks are given higher priority based on weights assigned to the number of incomplete write operations and available free disk space. This prioritization scheme avoids slow or busy disks while not overloading a particular disk.



(a) Randomized hashing provides better load balancing (μ = 609.22, σ = 52.67).



(b) Order-preserving hashing reduces metadata tree growth by ~81%.



(c) Order-preserving hashing does not balance loads (μ = 609.22, σ = 1063.84).

Fig. 6. Effect of consistent hashing and order-preserving hashing.

Figure 4 shows the ingestion rate, memory usage, and aging activities at a Gossamer node holding 859 entities. We ingested a stream of Spinneret (with probabilistic hash) instances consuming up to 85% of the available bandwidth. Aging helps maintain the overall memory consumption of the node below the upper threshold of 8 GB (66% of 12 GB total memory). Figure 5 shows the breakdown of the number of sketches present in the system over time. The in-memory sketch count was roughly a constant while the number of sketches aged out increases over time.

Gossamer can also limit disk usage by preferentially removing fine-grained sketches that were aged to disk. On-disk aging follows a similar approach to in-memory aging and starts by removing the finest-grained catalogs.

2.3.3 Storing Metadata. At each node, Gossamer maintains an index for each CSE, *the metadata tree*, forming a distributed index for each CSE. The unique feature-bin combinations (that are part of the metadata) included in Spinneret instances are used to create an inverted index for individual sketches for efficient querying. This index helps reduce the search space of a query in two ways:

(1) It allows tracking all feature-bin combinations that have ever occurred — this avoids exhaustive querying over all possible feature-bin combinations on a sketch.
(2) By pointing to sketches where a particular feature-bin combination has been observed, the index helps avoid exhaustive searches over all available sketches.

The metadata tree is organized as a trie (prefix tree) with pointers to the corresponding sketches placed at the leaf nodes. We use a radix tree, which is a space efficient trie implementation, where a vertex is merged with its parent if it is the only child. With the NOAA data (Section 2.0.1), we have observed up to ~46% space savings with a radix tree compared to a trie. Insert and query complexity for radix tree is $O(m)$, where m is the length of the search query (m = no. of features $\times$ length of the bin identifier). Figure 3d shows an example metadata tree with five feature-bin combinations: `0102`, `0110`, `0112`, `040A`, and `040C`.

Sketch pointers returned from a query reference sketches containing feature-bin combinations of interest. A sketch pointer has two components: temporal and entity information and location of the sketch within the Gossamer server pool. Encoding this metadata into a sketch pointer facilitates in-place filtering of sketches for temporal and entity-specific predicates during query evaluations.

As more Spinneret instances are ingested, the in-memory metadata managed at the server nodes continue to grow. The growth of the metadata tree can be attributed to two factors: ① unique feature-bin combinations that increase the vertex and edge count, and ② sketches accumulating over time adding more leaf nodes. We expect that in most practical deployments the number of feature-bin combinations should stabilize over time. The growth of the leaf node count is controlled by the aging process; a set of sketch pointers are replaced by a pointer to the summary sketch.

2.3.4 Organizing the Server Pool. The Gossamer server pool is designed to manage data from multiple CSEs and is organized as a distributed hash table (DHT). DHTs are robust, scalable systems for managing large networks of heterogeneous computing resources. The consistent hashing scheme that underpins DHTs offers excellent load balancing properties and incremental scalability where commodity hardware can be added incrementally to meet rising storage or processing demands. DHTs represent data items as $< key, value >$ pairs; the keys are generated by hashing metadata elements identifying the data, while the value is the data item to be stored. In Gossamer, the entity identifier is used as the key whereas the value can either be the sketch or the metadata. The Gossamer server pool is symmetric and decentralized; every Gossamer server has the same set of responsibilities as its peers, and there is no centralized control. This improves the system availability and scalability [21]. To reduce variability in sketch ingestion and query latency via efficient peer lookups, Gossamer uses $O(1)$ routing (zero-hop routing) [55].

Initially, we stored the sketches and metadata for a given entity at the Gossamer server responsible for *hash(entity id.)*. We performed a microbenchmark to assess this design choice. We distributed data corresponding to 60922 entities in the 2014 NOAA dataset (Section 2.0.1) across 100 machines. Using a randomized hashing function, as is typically used for consistent hashing, combined with virtual nodes [21, 64] provided excellent load balancing properties. As can be seen in Figure 6a, randomized placement of entities load balances storage of sketches but results in a rapid growth of the metadata tree. This is due to the high diversity of the feature-bin combinations of unrelated entities stored in a single node that reduces reusable paths within the metadata tree.

This motivated the question: Would an order-preserving hash function outperform a randomized hashing function? An order preserving hashing function f for keys in S is defined as; $\forall k_1, k_2 \in S$: if $k_1 < k_2$ then $f(k_1) < f(k_2)$ [47]. The entity identifiers should be generated systematically such that similar entities would be assigned numerically close identifiers. For instance, geohashes [46] can be used as an entity identifier for spatial data, where nearby locations share the same prefix (Geohash strings will subsequently be converted to numeric values identifying their position within the ring using a lookup table similar to Pearson hashing [53]). This results in a significant reduction in the metadata tree growth. For NOAA data, we observed an ~81% improvement in memory consumption as shown in Figure 6b. The downside of this approach is poor load balancing of sketches due to uneven distribution of keys as shown in Figure 6c (confirmed in the literature [33]). In summary, **using randomized hashing exhibits better load balancing properties whereas order preserving hashing significantly reduces metadata tree growth.**

To harness benefits from both these schemes, we created two virtual groups of nodes within the Gossamer server pool: data nodes (for storing the sketches) and metadata nodes (for storing metadata). Sketch payload and metadata included in Spinneret instances are split and stored separately on these two groups of nodes. Nodes in each of these groups form a separate ring and use a hashing scheme that is appropriate for the type of the data that they store; data nodes use randomized hashing, and metadata nodes use order preserving hashing. This also allows the two groups of nodes to be scaled independently; for instance, over time, there will be more additions to the data nodes group (assuming a less aggressive aging scheme) whereas the number of metadata nodes will grow at a comparatively slower rate. This approach increases the query latency due to the additional network hop introduced between the metadata and the sketches. It will be mostly reflected on the latencies when querying the memory resident sketches whereas for the aged out sketches, the difference will not be significant [13].

In our storage cluster, in-memory data structures such as catalogs and metadata trees are stored in a persistent write-ahead-log to to prevent data loss during node failures. We will support high-availability (with eventual consistency guarantees) via replication in our DHTs in future.

2.4 Data Explorations & Enabling Analytics (RQ-1, RQ-4)

Data exploration is a four-step process involving query evaluations and construction and materialization of the *Scaffold.* First, the user defines the data of interest by using a set of predicates for the features and temporal scopes. Second, the metadata node identifies sketches (and the data nodes where they are resident) where the feature-bin combinations occur. Third, the data nodes probe these sketches to retrieve information about the occurrence frequencies and construct tuples that comprise the Scaffold. Finally, the Scaffold is materialized to produce an exploratory dataset that is statistically representative, distributed to align with the expected processing, and represented as HDFS [8] files to support interoperation with analytical engines. Several analytical engines, such as Hadoop MapReduce, Spark, TensorFlow, Mahout, etc., support integration with HDFS (Hadoop Distributed File System) and use it as a primary source for accessing data. HDFS, which is data format neutral and suited for semi/unstructured data, thus provides an excellent avenue for us to interoperate with analytical engines. Most importantly, users can use/modify legacy code that they developed in their preferred analytical engines with the datasets generated from Gossamer.

2.4.1 Defining the Data of Interest. Data extraction is driven by predicates specified by the user through Gossamer's fluent style query API. These predicates enforce constraints on the data space for feature values, temporal characteristics, CSEs, and entities. For instance, a user may be interested in extracting data corresponding to cold days during summer for the last 5 years for Fort Collins (geohash prefix = `9xjq`) using NOAA data. The list of predicates attached to the query would be `cse_id == NOAA`, `entity_id starts with 9xjq`, `month >= June && month < Sept.`, `temperature < 277`, and `year >= 2013`. Queries can be submitted to any Gossamer node, which redirects them to Gossamer nodes holding metadata for matching entities.

In a public deployment, we expect to operate a registry in parallel to the storage cluster to manage metadata about the hosted datasets. The client will query the metadata registry during the query construction phase to explore dataset identifier(s), feature names and units of measurements. The registry can also be used to host bin configurations that need to be shared among federated edge devices as discussed in Section 2.1.1.

2.4.2 Identifying Sketches With Relevant Data. At a Gossamer metadata node, the data space defined by the feature predicates is first mapped to a series of feature-bin combination strings to be queried from the metadata tree. The feature predicates are evaluated in the same order as the feature values in observations were discretized into feature-bin vectors at the edges. If there is a predicate for a feature, the range of interest is mapped to the set of bins encompassing the range using the same bin configuration that was used at the edges. In cases where no predicate is specified

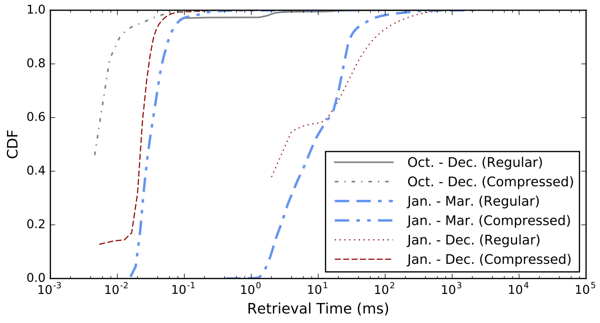


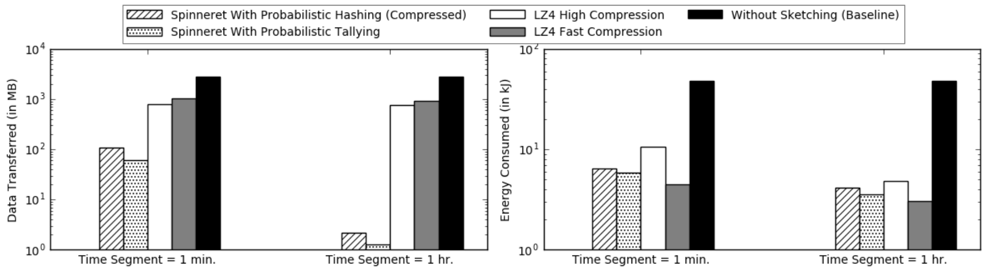
Fig. 7. Sketch retrieval times for different temporal scopes of the same query. Retrievals corresponding to the most recent data required fewer disk accesses.

for a feature, it is considered a wild card and the entire set of bins is considered. It is possible that the thresholds provided in the predicates do not perfectly align with the boundaries of the bins. In such cases, the thresholds are relaxed to match the closest bin encompassing the range specified in the predicate. For instance, for the temperature predicate in the above example (`temperature < 277`), if the bin boundaries surrounding the predicate threshold are `274.5` and `279.9`, then the predicate is relaxed to `279.9`. Construction of feature-bin combinations happens step-wise by iterating through features and their bins, gradually constructing a prefix list that eventually turns into the list of observed feature-bin combinations defined by the feature predicates. A new bin is appended to an existing feature-bin prefix in the set only if there an observed feature-bin combination starting with the new prefix. This is implemented using prefix lookups on the radix tree and reduces the search space significantly, especially when there are wild card features. Once the feature-bin strings are constructed, the radix tree is queried to retrieve the sketch pointers for each feature-bin combination. Temporal metadata embedded in sketch pointers (as explained in Section 2.3.3) is used to filter out sketches that do not satisfy the temporal bounds. The results of these queries are a set of tuples of the format ⟨*data node, sketch pointer, feature-bin combination*⟩.

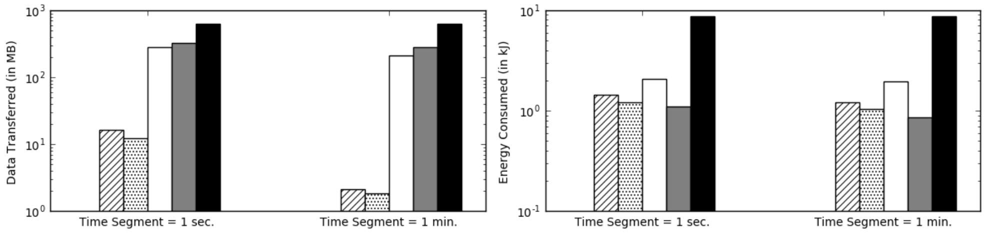
2.4.3 Constructing the Scaffold. A Scaffold is a distributed data structure constructed in response to a query and represents a portion of the data space. The list of sketches identified during query evaluations (Section 2.4.2) are probed at the data nodes to retrieve occurrence frequencies for the particular feature-bin combinations. A Scaffold comprises a set of tuples of the form ⟨*CSE Id., Entity Id., time segment, feature-bin combination, estimated frequency*⟩. Scaffolds are constructed in-place; tuples comprising the scaffold are retrieved and pinned in memory at the data nodes until being specifically discarded by the user. Gossamer also records gaps in time catalogs (due to missing sketches) within the temporal scope of the query while Scaffolds are constructed. Once constructed Scaffolds are reusable — they can be materialized in myriad ways to support exploratory analysis. Scaffolds can also be persisted on disk for later usage.

To conserve memory, in-place Scaffolds are compacted at each node. Given the repeated values for CSE and entity identifiers and feature-bin combination strings, we apply a lossless compression scheme (based on lookup tables) to the Scaffold during its construction. This scheme uses the same concept as Huffman coding [71] to provide an online compression algorithm that uses fixed-length codes instead of variable-length codes. After constructing local segments of the Scaffold, data nodes send an acknowledgment to the client; additional details include the number of feature-bin combinations, the number of observations, and gaps, if any, in the temporal scope. At this time, users can opt to download the Scaffold (provided enough disk space is available at the Driver), and inspect it manually before materializing as explained in Section 2.4.4.

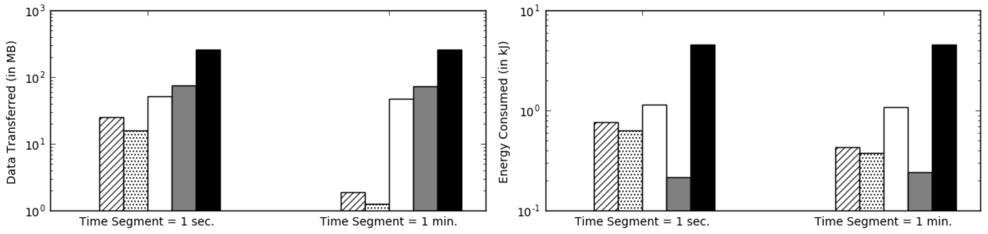
We performed a microbenchmark to evaluate the effectiveness of memory residency of the most relevant sketches. Under the default aging policy, Gossamer attempts to keep the most recent sketches in memory. We ingested the entire NOAA dataset for year 2014 and evaluated the same query for three different temporal scopes within 2014: January — December, January — March, and October — December. The results of this microbenchmark are depicted in Figure 7 for Spinneret with probabilistic hashing (compressed and regular). For the temporal scope corresponding to the most recent data (October — December), most of the relevant sketches are memory resident ($\sim 97\%$) resulting in lower retrieval times. All sketches for the temporal scope of January — March had been aged out, and these retrievals involved accessing disks. The annual temporal scope required accessing a mixture of in-memory ($\sim 15\%$) and on-disk sketches ($\sim 85\%$). The role of the disk cache is also evident in this benchmark. Due to the smaller storage footprint of the compressed sketch, the aged-out sketches are persisted into a few blobs that fit in the disk cache thus requiring fewer



(a) NOAA dataset (for two weeks): 10 features, 1 observation/s



(b) Gas sensor array under dynamic gas mixtures dataset: 18 features, 100 observations/s



(c) Smart home dataset: 12 features, 1000 observations/s

Fig. 8. Effectiveness of Spinneret at the edges with different frequency-based sketching algorithms and time segments with respect to data transfer and energy consumed. We compare Spinneret with binary compression scheme LZ4 under two compression configurations. We include the data transfer and energy consumption without any preprocessing as the baseline.

disk accesses during their retrieval. With regular sketches, the disk cache is not effective due to the large number of blobs and requires far more disk accesses.

2.4.4 Materialization. Materialization is the process of generating a dataset representing the data space of interest using the Scaffold as a blueprint. Upon constructing the Scaffold, a user may send a materialization request to all data nodes holding the Scaffold segments. A materialization request contains a set of directives including the number of data points required, sharding scheme, export mode, further refinements, and transformations on the feature values. A materialization operation begins by converting the feature-bin combinations back to feature values. By default, Gossamer uses the midpoint of the bin as the feature value but can be configured to use another value. This operation is followed by the refinements and transformations phase where the set of feature values are preprocessed as requested by users. For instance, users can choose a subset of features in the Scaffold to be present in the generated dataset, convert readings to a different unit of measurement, etc. The next phase is the data sharding phase where tuples in Scaffold segments are shuffled across the data nodes based on a key. This phase allows users to perform a *group by* operation

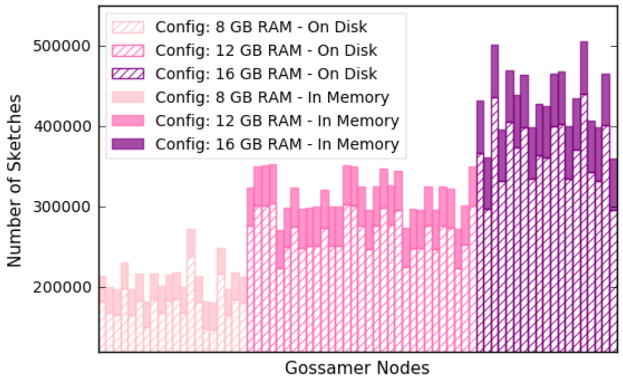


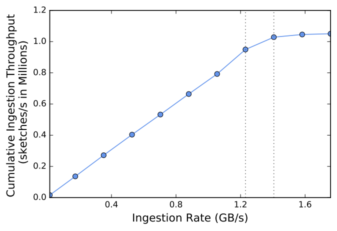
Fig. 9. Load distribution within the Gossamer data nodes while accounting for the node heterogeneity.

on the tuples of the generated dataset based on some attribute such as entity, feature value range, etc. Following the previous example, if the user wants to group the anomalous temperatures by month, the sharding attribute can be set to the month of the time segment. Sharded Scaffolds are encoded using the same compression scheme used when constructing the Scaffold, reducing network transfers (by at least 20% for 2014 NOAA data).

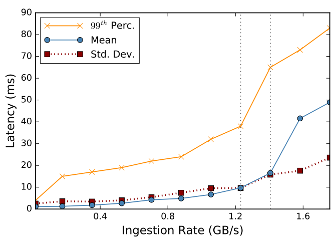
Once a data node receives all sharded Scaffolds from every other node, it starts generating the exploratory dataset. Using the total number of observations and the size of the required dataset, a Gossamer node determines the scaling factor (required dataset size/total observation count). Based on the scaling factor, a node either starts sampling (scaling factor < 1) or inflating (scaling factor ≥ 1). In addition to providing an extensible API, we support two built-in schemes to export exploratory datasets: export to HDFS or send as a stream to a provided endpoint. The generation and exporting of data happens in a streaming fashion where records are appended to the HDFS files (we create a separate file for every shard) or to the stream as they are generated. In both export modes, we append records as mini batches to improve the network I/O. The streaming appends allow us to maintain only a minimal set of generated data in-memory at a given time.

3 SYSTEM BENCHMARKS

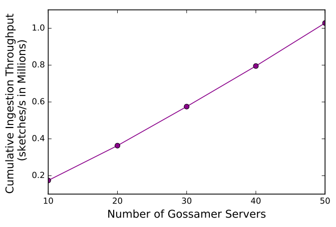
In this section, we evaluate how Gossamer improves ingestion (Section 3.2 and 3.4), storage (Section 3.3 and 3.4), and analytics (Section 3.5) of multi-feature streams originated at CSEs.



(a) Cumulative ingestion throughput vs. data ingestion rate (in a 50 node cluster).



(b) End-to-end ingestion latency vs. data ingestion rate (in a 50 node cluster).



(c) Cumulative ingestion throughput vs. cluster size (with 1.4 GB/s ingestion).

Fig. 10. Evaluating system scalability w.r.t. data ingestion.

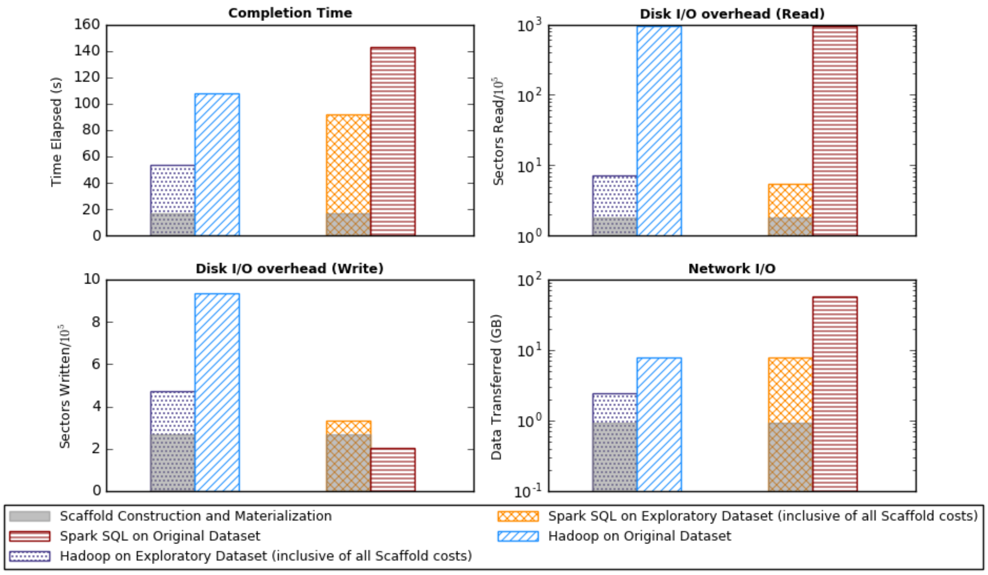


Fig. 11. Contrasting the costs of analytic jobs performed on exploratory datasets and original datasets.

3.1 Experimental Setup

3.1.1 Hardware and Software Setup. Performance evaluations reported here were carried out on a heterogeneous cluster comprising 30 HP DL320e servers (Xeon E3-1220 V2, 8 GB RAM), 48 HP DL160 servers (Xeon E5620, 12 GB RAM), and 100 HP DL60 servers (Xeon E5-2620, 16 GB RAM). A combination of these machines were used depending on the nature of the benchmark as explained in respective sections. The test cluster was configured to run Fedora 26 and Oracle Java runtime 1.8.0_65. We used a Raspberry Pi as the edge device and its power measurements were carried out using a Ubiquiti mFi mPower Mini smart plug. Analytic tasks were implemented using Apache Spark 2.0.1 [3], Scikit-learn 0.19.1 [54], and Apache HDFS 2.7.3 [8].

3.1.2 Datasets. We used three datasets from different domains for our experiments.

(1) *NOAA 2014 dataset* was our primary dataset as introduced in Section 2.0.1.
(2) *Gas sensor array under dynamic gas mixtures dataset* [24] includes time series data generated by 16 sensors when exposed to varying concentration levels of Ethylene and CO. The dataset contained 4,208,262 observations at a rate of 100 observations/s and 18 features.
(3) *Smart home dataset* from ACM DEBS 2014 grand challenge [1] containing power measurements (current active power and cumulative energy consumption) collected from smart plugs deployed in houses in Germany. We considered active power measurements from a single household

Table 1. Evaluating data ingestion to Amazon Web Services cloud in a multi-entity setup

Approach	Data Transferred (MB/Hour)	Energy Consumption (J/Hour)	Estimated Cost (USD/Year)
Spinneret (1-min/Probabilistic Hashing)	0.21	230.70	12
LZ4 High Compression	3.41	250.34	12
LZ4 Fast Compression	3.71	217.57	12
Without Sketching (Baseline)	5.54	1586.83	540

consisting of 12 plugs to construct an observational stream with 12 features, producing data at the rate of 1000 observations/s. The dataset encompasses 2,485,642 observations.

3.2 Edge Profiling (RQ-1, RQ-2)

We profiled the effectiveness of Spinneret at the edges of the network using based on two metrics: data transfer and the energy consumption. Spinneret was configured with the two types of sketching algorithms, probabilistic hashing and probabilistic tallying, with different time segment lengths. We compared its performance against the binary compression scheme LZ4. Binary compression was chosen instead of other data compression techniques designed for edges due to its support for multi-feature streams. With LZ4, the compression level can be configured — we used two compression levels in our benchmark. As the baseline, we included the data transfer and energy consumption results for traditional transmission without any preprocessing.

This benchmark was performed for a single entity in each of the datasets to simulate the data transmission and energy consumption at a single edge device. We expect the improvement we observe to linearly scale with time as well as the number of entities in the CSE. Due to the low frequency of observations in NOAA data, only for this particular benchmark we used cubic spline interpolation to impute intermediate data points (1 observations/s) as recommended in research literature for meteorological data [14, 42] and considered data for two weeks. Energy measurements that we report were inclusive of the processing and transmissions over MQTT.

Results are summarized in Figure 8. **We observed significant reductions in both the amount of data transferred (by a factor of ~26 - 2207 for the NOAA data, ~38 - 345 for the gas sensor array data, and ~10 - 203 for the smart home data) as well as in energy consumption (by a factor of ~7 - 13 for the NOAA data, ~6 - 8 for the gas sensor array data, and ~5 - 12 for the smart home data) when Spinneret is used.** These benchmarks substantiate our design assumption: reduced communication overhead outweighs the computational overhead associated with generating Spinneret sketches. Spinneret consistently outperforms both LZ4 configurations w.r.t. data transfer. LZ4 fast compression provides the lowest energy consumption across all schemes considered while Spinneret's energy consumption is comparable with the LZ4 fast compression. It should be noted that the Spinneret not only provides efficient data transfer, but also provides native support for efficient querying and aging after storage unlike compression schemes. In a real world deployment with unreliable networks, retransmissions are possible. In such cases, regular data ingestion (without any preprocessing) is susceptible for more communication errors contributing to higher energy consumption.

We extended the previous benchmark to include multiple entities and to ingest data into a commercial public cloud. We chose imputed NOAA data from 17 weather stations in northern Colorado spread across an area of $408km^2$. Data from each weather station was handled by a separate Raspberry Pi. We integrated Gossamer edge module with the Amazon IoT SDK [5] to ingest sketched data into an Amazon EC2 cluster. In this deployment, each Raspberry Pi was considered as a separate "AWS IoT Thing". We measured the volume of data transfer and energy consumption per month to ingest data from the 17 weather stations we considered as summarized in Table 1. **We were able to observe similar reductions in data transfer (~26×) and energy consumption (~6.9×) as with the benchmark with a single entity (Figure 8)**. Further, we estimated the annual data ingestion costs for each approach (assuming a monthly billing cycle). This calculation only considers the cost of data transfer and does not include other costs such as device connectivity costs. Because message sizes in all our approaches were within the limit enforced by Amazon IoT, the cost was primarily determined by the number of messages transferred in units of 1 million messages. **We were able to reduce the costs by 97.8% compared to regular data ingestion.** Even though the reduction in data volume does not affect the ingestion cost in this

Table 2. Descriptive statistics for original full-resolution data vs. exploratory data generated by Gossamer.

Feature ($Unit$)	Mean		Std. Dev.		Median		Kruskal-Wallis (P-Value)
	Original	Expl.	Original	Expl.	Original	Expl.	
Temperature (K)	281.83	281.83	13.27	13.32	281.39	281.55	0.83
Pressure (Pa)	83268.34	83271.39	5021.02	5047.81	83744.00	83363.23	0.81
Humidity (%)	57.50	57.49	22.68	22.68	58.0	56.70	0.80
Wind speed (m/s)	4.69	4.69	3.77	3.78	3.45	3.47	0.74
Precipitation (m)	11.44	11.45	7.39	7.45	9.25	8.64	0.75
Surf. visibility (m)	22764.18	22858.20	4700.16	4725.30	24224.19	24331.02	0.00

scenario, it directly affects the storage costs. Also it may contribute to increased data ingestion costs with other cloud providers such as Google Cloud where ingestions costs are calculated based on the volume of data transfer [12].

3.3 Load Balancing (RQ-1, RQ-3)

Gossamer attempts to distribute the load evenly across the servers while accounting for their heterogeneity. Figure 9 depicts the snapshot of the distribution of sketches after ingesting the entire 2014 NOAA dataset with a breakdown of the in-memory and on-disk (aged) sketches. We use the memory capacity of a server as the primary factor that determines its capability, because a node with larger memory can maintain a higher number of in-memory sketches and larger segments of Scaffolds. By adjusting the number of virtual nodes allocated to a server, Gossamer places more sketches on servers with better capabilities.

3.4 Scalability of Gossamer (RQ-1, RQ-3)

We evaluated the scalability of Gossamer with respect to data ingestion. In the first phase of the benchmark, we used a fixed cluster size (50 nodes: 35 data nodes and 15 metadata nodes) and increased the data ingestion rate while measuring the cumulative ingestion throughput across the cluster. A data node can support a higher ingestion throughput initially when all sketches are memory resident. But over time, when the memory allocated for storing sketches is depleted and the aging process is triggered, the maximum ingestion rate is bounded by throughput of the aging process (i.e., number of sketches that can be aged out to the disk in a unit period of time). The performance tipping point (1.04 million sketches/s) for the system was reached when the data ingestion rate was increased up to 1.2 — 1.4 GB/s as shown in Figure 10a. In other words, **a Gossamer cluster of 50 nodes can support 1.04 million edge devices each producing a sketch every second**.

As depicted in Figure 10b, after the system reaches the maximum possible ingestion throughput, there is a significant increase in the mean latency due to the queueing delay.

In the second phase of this benchmark, we maintained a constant data ingestion rate (1.4 GB/s) and increased the number of Gossamer servers. We maintained a ratio of 7:3 between the data nodes and the metadata nodes throughout all configurations. The cumulative ingestion throughput linearly increases with the number of servers as shown in Figure 10c. This demonstrates that the server pool organization scales as the number of available nodes increases.

3.5 Reducing the Costs of Analytic Jobs (RQ-1, RQ-4)

This experiment evaluates the effectiveness of Gossamer's data extraction process and how the exploratory datasets can reduce the costs of running analytical tasks. Our use case is to generate histograms of temperature, pressure and humidity for each month in summer 2014 in Colorado. First, a Scaffold is constructed for data from Colorado weather stations for the duration from June 21 to Sept. 22 in 2014. Scaffold materialization comprises three major steps: extracting the features of interest (temperature, pressure, and humidity), shard based on the month of the day, and export the exploratory dataset into HDFS. An analytical job was executed on this filtered and sharded dataset and contrasted with the job that ran on the original full resolution data. We used Hadoop and Spark SQL to execute the analytical tasks and measured job completion time, disk I/O overhead, and network I/O. The measurements for running analytical jobs on exploratory datasets include the costs of Scaffold construction, materialization, and running the job. Sketches corresponding to these analytical tasks were stored on disk to provide a fair comparison.

The results of this benchmark are depicted in Figure 11 including the measurements for Scaffold construction and materialization. **We see a major improvement in number of disk reads: 99.2% and 99.4% for Hadoop and Spark SQL respectively.** This improvement is mainly due to the efficient construction of the Scaffold; accessing only relevant portions of the data space using the metadata tree and the sketch based compact storage of data within data nodes. Low number of input splits in the exploratory dataset resulting in a low number of shuffle writes and the compact wire formats used by Gossamer contribute to its **lower network footprint: 68.4% and 86.3% improvements for Hadoop and Spark SQL respectively.** We observed an increased number of disk writes when Gossamer was used compared Spark SQL; the majority of the disk writes performed when Gossamer was used corresponds to writing the exploratory dataset into HDFS, which outnumbers the local disk writes performed by Spark SQL. **Overall, we observed up to 50.0% improvement in job completion times with Gossamer exploratory datasets.** We expect to see even more improvements when (1) we reuse scaffolds, (2) the data space encompasses in-memory sketches, and (3) the data space of interest is smaller.

4 ANALYTIC TASKS

Here, we evaluate the suitability of the exploratory datasets produced by Gossamer for real-world analytical tasks.

Dataset and Experimental Setup: We considered three specific regions from the 2014 NOAA data in Florida, USA (geohash: `f4du`), Hudson Bay, Canada (geohash: `djjs`), and Colorado, USA (geohash: `9xjv`). We mainly used following features: temperature, pressure, humidity, wind speed, precipitation, and surface visibility. For certain analytical tasks, additional refinements were applied during the materialization to extract a subset of the features, and the observation timestamps were considered as another feature. The size of the exploratory dataset was set to the size of the original dataset. We optimized analytical tasks for the original full-resolution data and used the same parameters when performing analytics using exploratory datasets.

4.1 Descriptive Statistics

The objective of this benchmark is to contrast how well the statistical properties of the original data are preserved by Gossamer in the presence of discretization and sketching. We compared the descriptive statistics of the exploratory dataset generated by Gossamer with that of the original full-resolution data. The results of this comparison is summarized in Table 2. We have only included the mean, standard deviation, and median due to space constraints. Statistics of the exploratory dataset do not significantly deviate from their counterparts in the original dataset.

Further, we performed a Kruskal-Wallis one-way analysis of variance test [35] to check if there is a significant statistical difference between the two datasets. This test provides a non-parametric approach to compare samples and validate if they are sampled from the same distribution. In our tests, except for the surface visibility feature, every other feature reported a $p - value$ higher than any widely used significance level. There was not enough evidence to reject the null hypothesis — the medians of the two populations are equal. For surface visibility, we saw a skew of values towards the higher end as depicted by Figure 12. If we consider only the lower values (< 23903.30), there is no significant statistical difference between the two datasets ($p - value$ for the Kruskal-Wallis test is 0.87). Because the majority of the values are placed in a single bin by the discretization process, the variability of the data at the higher end is lost, which accounts for more than 87% of the dataset (std. dev for original data - 19.84, Gossamer exploratory data - 0.00). This causes the Kruskal-Wallis test to report a significant statistical difference for surface visibility. Situations like this can be avoided through careful assignment of bins.

4.2 Pair-wise Feature Correlations

We calculated feature-wise correlations for both datasets separately using the Pearson product-moment correlation coefficients. We did not observe (Figure 13) any major deviations between cells in the two correlation matrices.

4.3 Time-Series Prediction

We assessed the suitability of using exploratory datasets to train time-series models. We trained a model using ARIMA [16] to predict the temperatures for an entity in Ocala, Florida (geohash: `djjumg29n`) for the month of March. We used data for the first 22 days in March to train the model and tried to predict temperatures for the next 7 days. With imputed high-frequency observations (as explained in Section 3.2), we could build a time-series model with high accuracy (RMSE = 0.0005 (K)) from the original full-resolution data. We used segment sizes of 1 hour when ingesting the interpolated data into Gossamer. Given that we cannot guarantee the ordering between observations within a segment, we used the average temperature observed within a segment during the exploratory dataset generation. So we used less frequent observations (1 obs./hr) when building the time-series model with exploratory data.

The same auto-regressive, difference, and moving average parameters determined for the ARIMA model (p, d, q) for the original full-resolution data were used to build the time-series model with the exploratory dataset generated by Gossamer. Predictions from both time-series models were contrasted as depicted in Figure 14. The time-series model generated by the exploratory data predicts the temperature within a reasonable offset from predictions generated based on the original full-resolution data (maximum difference between predictions is 1.59%, RMSE = 1.78 (K)).

4.4 Training Regression Models

We also contrasted the performance of Gossamer when constructing regression models. We used Spark Mllib to train regression models based on Random Forest to predict temperatures using

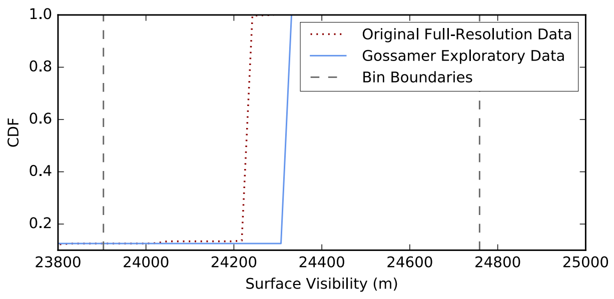


Fig. 12. Cumulative distribution function for surface visibility. Values are skewed towards the higher end.

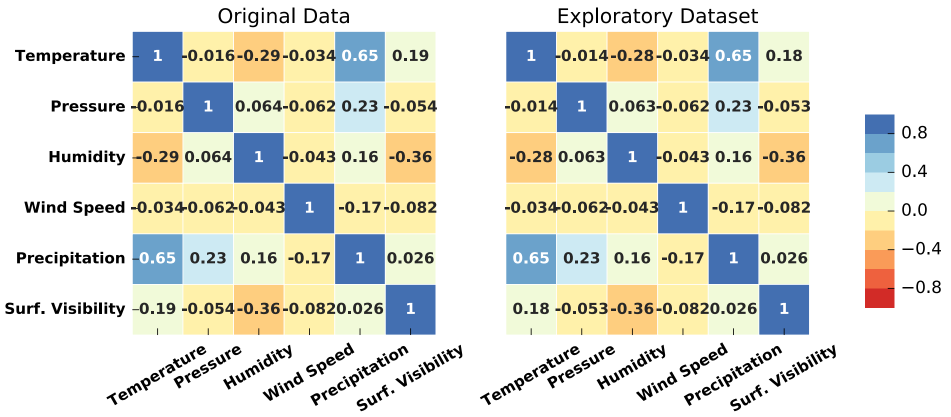


Fig. 13. Feature-wise correlations for original full-resolution data and exploratory dataset.

Original Data:

	Temperature	Pressure	Humidity	Wind Speed	Precipitation	Surf. Visibility
Temperature	1	-0.016	-0.29	-0.034	0.65	0.19
Pressure	-0.016	1	0.064	-0.062	0.23	-0.054
Humidity	-0.29	0.064	1	-0.043	0.16	-0.36
Wind Speed	-0.034	-0.062	-0.043	1	-0.17	-0.082
Precipitation	0.65	0.23	0.16	-0.17	1	0.026
Surf. Visibility	0.19	-0.054	-0.36	-0.082	0.026	1

Exploratory Dataset:

	Temperature	Pressure	Humidity	Wind Speed	Precipitation	Surf. Visibility
Temperature	1	-0.014	-0.28	-0.034	0.65	0.18
Pressure	-0.014	1	0.063	-0.062	0.23	-0.053
Humidity	-0.28	0.063	1	-0.043	0.16	-0.36
Wind Speed	-0.034	-0.062	-0.043	1	-0.17	-0.082
Precipitation	0.65	0.23	0.16	-0.17	1	0.026
Surf. Visibility	0.18	-0.053	-0.36	-0.082	0.026	1

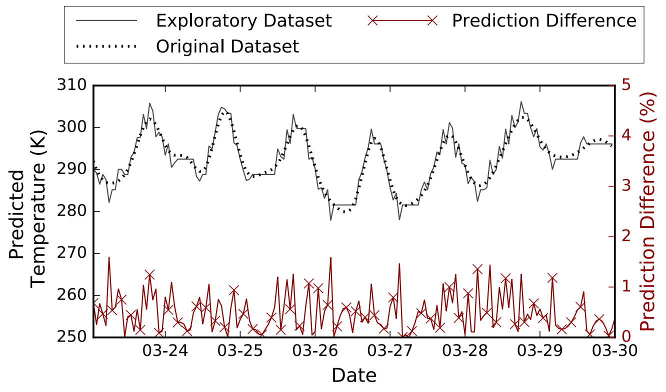


Fig. 14. ARIMA predictions for temperature.

surface visibility, humidity, and precipitation for each of the three regions. Similar to previous analytical tasks, parameters used for building the model (number of trees, maximum depth of the trees, and maximum number of bins) were optimized for the original full-resolution data, and the same parameters were used with the exploratory dataset. The accuracy of these models was measured using a test dataset extracted from the original full-resolution data (30%). As reported in Table 3, the predictions generated by the models trained using the two datasets are quite similar.

5 RELATED WORK

Data Reduction at the Edges: We discuss edge mining, sampling, and compression where data streams are preprocessed at the edges looking for repeating patterns, similarity between consecutive observations, and other properties useful in compacting the data streams at the edges.

Edge mining techniques [17, 22, 27, 29, 63, 72], used in the context of wireless sensor networks, focus on processing the data stream locally to summarize them into a compact derived stream such as stream of state changes or aggregates. This approach effectively reduces the number of messages that needs to be transferred for further processing. In CAROMM [63], observational streams are dynamically clustered at the edge devices, and only the changes captured in the sensed environment are transferred to the cloud. Gaura et al. [27] propose an algorithm that captures the time spent in various states using time-discounted histogram encoding algorithm instead of transferring individual events. For instance, instead of reporting the stream of raw metrics provided by a gyroscope, this algorithm can process the stream locally to calculate the time spent on

Table 3. Constrasting performance of two models trained with the full-resolution data and exploratory data.

Region	Avg. Temp. (K)	RMSE - Original(K)		RMSE - Exploratory(K)	
		Mean	Std. Dev.	Mean	Std. Dev.
`djjs`	265.58	2.39	0.07	2.86	0.05
`f4du`	295.31	5.21	0.09	5.01	0.09
`9xjv`	282.11	8.21	0.02	8.31	0.02

various postures by a human subject. While providing efficient reductions in data transfer between the sensing and processing layers, the edge mining techniques are tightly coupled with current application requirements. On the other hand, Spinneret sketches are a compact representations of the raw stream itself; and caters to a broader set of future application requirements.

Sampling is effective in most CSEs where features do not demonstrate randomized behaviors. AdaM [68] is an adaptive sampling algorithm, which adjusts the sampling interval based on the variability of the observed feature. A stream is considered stable if the estimated standard deviation approximates the observed standard deviation of the feature values with high confidence — in such cases the sampling rate is lowered. Traub et al. [67] propose a scheme based on user-defined sampling functions to reduce data transfers by fusing multiple read requests into a single sensor read. User-defined sampling functions provide a tolerance interval declaring an acceptable time interval to perform the sensor read. The read scheduler tries to fuse multiple read requests with overlapping tolerance intervals to identify the best point in time to perform the sensor read. This works well in stream processing settings where current queries govern ongoing data transfers; while, this approach is limiting in settings where future analytic tasks depend on historical data. Furthermore, these sampling schemes are primarily designed for single feature streams (some schemes require to pick a primary feature in the case of multi-feature streams to drive the sampling [67]) whereas Spinneret is designed for multi-feature streams.

Compression also leverages the low entropy of the observational streams. Most lossless compression algorithms [39, 49, 58] designed for low powered edge devices use dictionary based lookup tables. Lossless Entropy Encoding (LEC) [39] uses a dictionary of Huffman codes. An observational stream is modeled as a series of differences — an observation is replaced by its difference from the preceding observation. The differences are encoded by referencing a dictionary of Huffman codes, where the most frequently encountered values are represented using shorter bit strings. This operates on the assumption that consecutive observations do not deviate much from each other. LTC [61] leverages the linear temporal trends in data to provide lossy compression. These schemes are designed to compress single-feature streams whereas Spinneret can generate compact representations of the multi-feature streams. Further, the Spinneret instances can be queried without materialization which is not feasible with compression-based schemes.

Edge Processing: Edge processing modules are prevalent in present data analytics domain [2, 6]. They provide general purpose computation, communication, and device management capabilities within a lightweight execution runtime which can be leveraged to build data ingestion and processing pipelines. For instance, Amazon's Greengrass modules deployed at an edge device can work with a group of devices (data sources) in close proximity to provide local data processing, communications with the cloud for further processing and persistent storage, and data caching. Similarly, Apache Edgent provides a functional API to connect to streaming sources and process observations. We expect an excellent synergy between the Gossamer's edge processing functionality and the capabilities offered by these modules. The Gossamer edge processing functionality can be implemented using the processing and communication APIs provided by these edge modules.

Time-Series Data Storage: Storage solutions specifically designed for time-series data [7, 9–11] are gaining traction recently. Prometheus [11] and Graphite [7] are designed for monitoring — storing numeric time-series data for various metrics, visualizations, and alerting are supported. There is a fundamental difference in Gossamer and these databases — Gossamer can be viewed as an observation(or event) based storage system whereas Prometheus and Graphite are designed to store metrics derived from events. Also these efforts do not offer any native aging scheme. InfluxDB is designed for storing events and supports retention policies to control the growth of the data through downsampling. Using retention policies, users can generate and store an aggregate (e.g. mean) of the older data instead of the raw data. Gossamer's aging policy is more flexible than that of InfluxDB's — a summary of the observed frequencies is kept at a coarser granularity instead of aggregates therefore not restricting the future analytics on aged data.

There are two primary difference between all these engines and Gossamer. (1) Their query model closely follows the SQL model where users query the database for specific answers. In Gossamer, queries are used to extract a portion of the data space for further analysis using analytical engines. (2) Gossamer provides a unified data model based on Spinneret for both ingestion and storage. Time-series databases usually depend on another system for data ingestion and often involves a data transformation step before the storage.

Distributed Sketching: Sketches have been used to store the observational streams in a compact format [18, 65] at the center. Synopsis [18] proposes a memory-resident distributed sketch organized as a prefix-tree for spatio-temporal data. Gossamer uses sketching in a different way compared to Synopsis; an ensemble of sketches is used to store the temporal segments of observational streams in a compact form instead of an all-encompassing sketch. Using the statistical information maintained within summaries, Synopsis can support richer queries such as with predicates on correlation between features. Tao et al. [65] support distinct count queries over spatio-temporal data using the FM algorithm [23] to estimate the number of distinct objects in a dataset. Sketches are organized using a combination of an R-tree and B-tree. The R-tree is used to index different regions, and the leaf nodes point to a B-tree, which stores the historical sketches for that particular region. This scheme eliminates the need for storing individual elements through the use of sketches, hence reduces the space overhead considerably. However, it is designed for single-feature observational streams with restrictive query types.

The use of the aforementioned systems is predicated on using a spatial attribute as one of the required features of the stream — this is not required in Gossamer. Also both systems target compact storage of data at the center and do not reduce data at the edges where as Gossamer provides an end-to-end solution encompassing both efficient ingestion and storage of data streams.

Distributed Queries: Leveraging fog for federated query processing has been studied in [32, 38] where the data dispersed between edge devices and cloud can be queried. In Hermes [38], most recent data is stored on edge devices organized as a hierarchy of reservoir samples for different time granularities. Older data is pushed to the cloud, and cloud nodes coordinate query evaluation between cloud and edge devices. Spinneret complements these methodologies — sampled data can be sketched to achieve further compactions during the storage both at the edges and the center.

Harnessing capabilities of edge devices for distributed stream processing has been gaining traction. PATH2iot [41] attempts to distribute stream processing queries throughout the entire infrastructure, including edge devices, optimizing for non functional requirements such as energy consumption of sensors. Renart et al. [56] propose a content based publisher-subscriber model to connect data consumers with proximate data producers and computing resources at to provide location-aware services. These systems are designed with a different objective: providing support for context-aware, low-latency queries and data processing. Sensor network platforms often support dimensionality reduction and distributed queries [28, 37]. In general, these systems are designed around edge devices that are much less capable than those in modern IoT deployments, and therefore have limited processing/analysis responsibilities. DIMENSION [25] leverages wavelets for efficient, lossy compression of readings from devices such as Crossbow Motes and uses hierarchical QuadTree-based routing to distribute queries. Systems like DIMENSION generally produce accurate data representations and support queries but are designed for lower-frequency data arrivals (1 sample/minute) and smaller datasets (~8 GB/year) than Gossamer.

6 CONCLUSIONS AND FUTURE WORK

In this study, we described our methodology for data management and analytics over CSE data.

RQ-1: Effective generation of sketches allows us to preserve representativeness of the data space while significantly reducing the ingestions overheads relating to energy and bandwidth utilization. Organizing the Gossamer server pool as a DHT and leveraging consistent hashing allows us to balance the storage workloads. through targeted memory-residency of these sketches, we reduce memory pressure and disk I/O and ensure faster retrievals and construction of exploratory datasets.

RQ-2: Data reduction is most effective when it is closest to the source; so data is sketched at the edges. Discretization and frequency based sketches allow data volume reductions across multidimensional observations and keeping pace with data arrival rates at the edges. Specifically, we *reduce* (1) data volumes transmitted from the edges accruing energy savings, (2) utilization and contention over the links, and (3) storage requirements at the servers. Using an ensemble of sketches preserves representativeness of data and ensures the usability for future application needs.

RQ-3: Effective dispersion, management, and organization of metadata underpins query evaluations. Using order-preserving hashes for distribution of metadata collocates similar metadata reducing memory footprint at each node. Organizing the metadata graph as a radix tree conserves memory. Our aging scheme implemented through sketch aggregation preserves the representativeness of aged data at coarser grained temporal scopes to control the growth of streaming datasets.

RQ-4: Materializing the exploratory dataset in HDFS allows us to interoperate with several analytical engines. To ensure timeliness, the materialization is aligned with the shards over which the processing will be performed. By constructing Scaffolds in-memory and in-place, disk accesses prior to materialization in HDFS are significantly reduced.

As part of future work, we will improve our fault tolerance guarantees and dynamic item balancing schemes [26, 33] where the positions of the nodes in the ring are adjusted during the runtime to improve load balancing in metadata nodes. Another avenue is a sketch-aligned, memory-resident HDFS to eliminate disk I/O during materialization.

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] 2014. DEBS 2014 Grand Challenge: Smart homes. http://debs.org/debs-2014-smart-homes/
[2] 2016. Apache Edgent: A Community for Accelerating Analytics at the Edge. http://edgent.apache.org/
[3] 2016. Apache Spark: Lightning-fast cluster computing. http://spark.apache.org
[4] 2018. Apache Hadoop: Open-source software for reliable, scalable, distributed computing. https://hadoop.apache.org/
[5] 2019. AWS IoT Core. https://aws.amazon.com/iot-core/
[6] 2019. AWS IoT Greengrass. https://aws.amazon.com/greengrass/
[7] 2019. Graphite. https://graphiteapp.org/
[8] 2019. HDFS Architecture. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[9] 2019. InfluxDB: The modern engine for Metrics and Events. https://www.influxdata.com/
[10] 2019. Open TSDB: The Scalable Time Series Database. http://opentsdb.net/
[11] 2019. Prometheus: From metrics to insight. https://prometheus.io/
[12] 2020. Cloud IoT Core. https://cloud.google.com/iot-core/
[13] Ganesh Ananthanarayanan, et al. 2011. Disk-Locality in Datacenter Computing Considered Irrelevant.. In *HotOS*, Vol. 13. 12–12.
[14] Juan-Carlos Baltazar et al. 2006. Study of cubic splines and Fourier series as interpolation techniques for filling in short periods of missing building energy use and weather data. *Journal of Solar Energy Engineering* 128, 2 (2006), 226–230.
[15] Flavio Bonomi, et al. 2012. 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, 13–16.
[16] George EP Box, et al. 2015. *Time series analysis: forecasting and control.* John Wiley & Sons.
[17] James Brusey, et al. 2009. Postural activity monitoring for increasing safety in bomb disposal missions. *Measurement Science and Technology* 20, 7 (2009), 075204.
[18] Thilina Buddhika, et al. 2017. Synopsis: A Distributed Sketch over Voluminous Spatiotemporal Observational Streams. *IEEE Transactions on Knowledge and Data Engineering* 29, 11 (2017), 2552–2566.
[19] Graham Cormode. 2011. Sketch techniques for approximate query processing. *Foundations and Trends in Databases. NOW publishers* (2011).
[20] Graham Cormode et al. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.
[21] Giuseppe DeCandia, et al. 2007. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.
[22] Pavan Edara, et al. 2008. Asynchronous in-network prediction: Efficient aggregation in sensor networks. *ACM Transactions on Sensor Networks (TOSN)* 4, 4 (2008), 25.
[23] Philippe Flajolet et al. 1985. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences* 31, 2 (1985), 182–209.
[24] Jordi Fonollosa, et al. 2015. Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring. *Sensors and Actuators B: Chemical* 215 (2015), 618–629.
[25] Deepak Ganesan, et al. 2005. Multiresolution storage and search in sensor networks. *ACM Transactions on Storage (TOS)* 1, 3 (2005), 277–315.
[26] Prasanna Ganesan, et al. 2004. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment, 444–455.
[27] Elena I Gaura, et al. 2011. Bare necessitiesâĂŤKnowledge-driven WSN design. In *SENSORS, 2011 IEEE*. IEEE, 66–70.
[28] Phillip B Gibbons, et al. 2003. Irisnet: An architecture for a worldwide sensor web. *IEEE pervasive computing* 2, 4 (2003), 22–33.
[29] Daniel Goldsmith et al. 2010. The Spanish Inquisition ProtocolâĂŤmodel based transmission reduction for wireless sensor networks. In *SENSORS, 2010 IEEE*. IEEE, 2043–2048.
[30] Patrick Hunt, et al. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems.. In *USENIX annual technical conference*, Vol. 8. Boston, MA, USA, 9.
[31] Yahoo Inc. 2017. Frequent Items Sketches Overview. https://datasketches.github.io/docs/FrequentItems/FrequentItemsOverview.html
[32] Prem Jayaraman, et al. 2014. Cardap: A scalable energy-efficient context aware distributed mobile data analytics platform for the fog. In *East European Conference on Advances in Databases and Information Systems*. Springer, 192–206.
[33] David R Karger et al. 2004. Simple efficient load balancing algorithms for peer-to-peer systems. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 36–43.
[34] Martin Kleppmann. 2017. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems.* " O'Reilly Media, Inc.".
[35] William H Kruskal et al. 1952. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association* 47, 260 (1952), 583–621.
[36] Dave Locke. 2010. Mq telemetry transport (mqtt) v3. 1 protocol specification. *IBM developerWorks* (2010).
[37] Samuel R Madden, et al. 2005. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)* 30, 1 (2005), 122–173.
[38] Matthew Malensek, et al. 2017. HERMES: Federating Fog and Cloud Domains to Support Query Evaluations in Continuous Sensing Environments. *IEEE Cloud Computing* 4, 2 (2017), 54–62.
[39] Francesco Marcelloni et al. 2009. An efficient lossless compression algorithm for tiny nodes of monitoring wireless sensor networks. *Comput. J.* 52, 8 (2009), 969–987.
[40] Massachusetts Department of Transportation. 2017. MassDOT developers' data sources. https://www.mass.gov/massdot-developers-data-sources

[41] Peter Michalák et al. 2017. PATH2iot: A Holistic, Distributed Stream Processing System. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 25–32.

[42] Walter F Miller. 1990. *Short-Term Hourly Temperature Interpolation.* Technical Report. AIR FORCE ENVIRONMENTAL TECHNICAL APPLICATIONS CENTER SCOTT AFB IL.

[43] Jayadev Misra et al. 1982. Finding repeated elements. *Science of computer programming* 2, 2 (1982), 143–152.

[44] National Oceanic and Atmospheric Administration. 2016. The North American Mesoscale Forecast System. http://www.emc.ncep.noaa.gov/index.php?branch=NAM

[45] Aileen Nielsen. 2019. *Practial Time Series Analysis.* " O'Reilly Media, Inc.".

[46] Gustavo Niemeyer. 2008. Geohash. http://en.wikipedia.org/wiki/Geohash

[47] NIST. 2009. order-preserving minimal perfect hashing. https://xlinux.nist.gov/dads/HTML/orderPreservMinPerfectHash.html

[48] Shadi A Noghabi, et al. 2016. Ambry: LinkedIn's Scalable Geo-Distributed Object Store. In *Proceedings of the 2016 International Conference on Management of Data.* ACM, 253–265.

[49] MFXJ Oberhumer. [n. d.]. miniLZO: mini version of the LZO real-time data compression library. http://www.oberhumer.com/opensource/lzo/

[50] Prashant Pandey, et al. 2017. A General-Purpose Counting Filter: Making Every Bit Count. In *Proceedings of the 2017 ACM International Conference on Management of Data.* ACM, 775–787.

[51] Apostolos Papageorgiou, et al. 2015. Reconstructability-aware filtering and forwarding of time series data in internet-of-things architectures. In *Big Data (BigData Congress), 2015 IEEE International Congress on.* IEEE, 576–583.

[52] Emanuel Parzen. 1962. On estimation of a probability density function and mode. *The annals of mathematical statistics* 33, 3 (1962), 1065–1076.

[53] Peter K Pearson. 1990. Fast hashing of variable-length text strings. *Commun. ACM* 33, 6 (1990), 677–680.

[54] F. Pedregosa, et al. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[55] Venugopalan Ramasubramanian et al. 2004. Beehive: O (1) Lookup Performance for Power-Law Query Distributions in Peer-to-Peer Overlays.. In *Nsdi*, Vol. 4. 8–8.

[56] Eduard Gibert Renart, et al. 2017. Data-driven stream processing at the edge. In *Fog and Edge Computing (ICFEC), 2017 IEEE 1st International Conference on.* IEEE, 31–40.

[57] Mathew Ryden, et al. 2014. Nebula: Distributed edge cloud for data intensive computing. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on.* IEEE, 57–66.

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

[59] Hooman Peiro Sajjad, et al. 2016. Spanedge: Towards unifying stream processing over central and near-the-edge data centers. In *2016 IEEE/ACM Symposium on Edge Computing (SEC).* IEEE, 168–178.

[60] M. Satyanarayanan, et al. 2009. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing* 4 (2009), 14–23.

[61] Tom Schoellhammer, et al. 2004. Lightweight temporal compression of microclimate datasets. (2004).

[62] Zach Shelby, et al. 2014. The constrained application protocol (CoAP). (2014).

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

[64] Ion Stoica, et al. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review* 31, 4 (2001), 149–160.

[65] Yufei Tao, et al. 2004. Spatio-temporal aggregation using sketches. In *Data Engineering, 2004. Proceedings. 20th International Conference on.* IEEE, 214–225.

[66] Bart Theeten et al. 2015. Chive: Bandwidth optimized continuous querying in distributed clouds. *IEEE Transactions on cloud computing* 3, 2 (2015), 219–232.

[67] Jonas Traub, et al. 2017. Optimized on-demand data streaming from sensor nodes. In *Proceedings of the 2017 Symposium on Cloud Computing.* ACM, 586–597.

[68] Demetris Trihinas, et al. 2015. AdaM: An adaptive monitoring framework for sampling and filtering on IoT devices. In *Big Data (Big Data), 2015 IEEE International Conference on.* IEEE, 717–726.

[69] Chun-Wei Tsai, et al. 2014. Data mining for Internet of Things: A survey. *IEEE Communications Surveys and Tutorials* 16, 1 (2014), 77–97.

[70] US Environmental Protection Agency. 2018. Daily Summary Data - Criteria Gases. https://aqs.epa.gov/aqsweb/airdata/download_files.html#Daily

[71] Jan Van Leeuwen. 1976. On the Construction of Huffman Trees.. In *ICALP.* 382–410.

[72] Chi Yang, et al. 2011. Transmission reduction based on order compression of compound aggregate data over wireless sensor networks. In *Pervasive Computing and Applications (ICPCA), 2011 6th International Conference on.* IEEE, 335–342.