Concept-Driven Load Shedding: Reducing Size and Error of Voluminous and Variable Data Streams

Nikos R. Katsipoulakis, Alexandros Labrinidis, Panos K. Chrysanthis
Department of Computer Science
University of Pittsburgh, Pittsburgh, PA
Email: {katsip, labrinid, panos}@cs.pitt.edu

***Abstract*—Load shedding is a technique that aims to ameliorate the consequences of the *Velocity* and the *Volume* of Big Data stream processing. When temporal input spikes appear, tuples are shed until a Stream Processing Engine's (SPE) processing capacity is not overwhelmed and results are produced in a timely fashion. Existing load shedding techniques have become obsolete and are not applicable to modern use-cases which require the extraction of patterns from continuously evolving (i.e., *Variable*) voluminous streams.**

In this work, we identify the shortcomings of existing load shedding techniques when applied to streams with *concept drift*. We propose Concept-Driven load shedding (CoD), which aims at limiting the data volume imposed on the SPE while producing high accuracy results. On top of that, we designed CoD for modern SPEs and made its overhead negligible. Our experiments indicate that CoD can deliver more than 10x more accurate results compared to the state of the art in load shedding. Also, CoD can offer up to 2.25x better performance compared to normal processing and reduce the processed data volume significantly.

I. Introduction

Real-time data analysis is becoming prevalent in various sectors, which depend on online pattern identification. Some examples include social network analysis, targeted advertising, click-stream analysis, urban analytics, etc. In order to cope with the continuous and voluminous data production, stream processing has been deemed as the most suitable processing model, since it requires a single pass over stream data as they arrive.

Stream Processing Engines (SPEs) [1]–[20] have been developed to meet the aforementioned requirements. SPEs receive a continuous query (CQ), having data from one or more data sources, and an output destination. Then, their responsibility is to produce results that match the CQ until they are requested to stop. A typical use-case for SPEs is when users require real-time insights for online decision making. For instance, an analyst in a ride-sharing company might be interested in the latest trending routes in a city. Trends will help the analyst adjust operational plans in order to decrease costs and increase revenue. Such adjustments could be manual or automated, through a machine learning module the analyst designed. Towards this, she could utilize the CQ shown in Fig. 1 (in functional notation); the output of this CQ could be the input to the automated decision module.

According to this query, tuples are coming from the *rides* input stream, and each one is assigned a timestamp, which comes from its own *time* field. In turn, this timestamp is used to assign each tuple to the time windows that it belongs in (with sliding windows each tuple participates in more than one window). For each window, the ten most profitable routes are extracted, by calculating the mean fare value for each route, ordering tuples based on the mean fare, and retrieving the ten routes with the highest average fares.

```
query = rides
     .time(x -> x.time)
     .windowSize(15, MINUTES)
     .windowSlide(5, MINUTES)
     .mean(x -> x.fare)
     .group(x -> x.route)
     .order()
     .limit(10);
```

Fig. 1: Motivating Example Query.

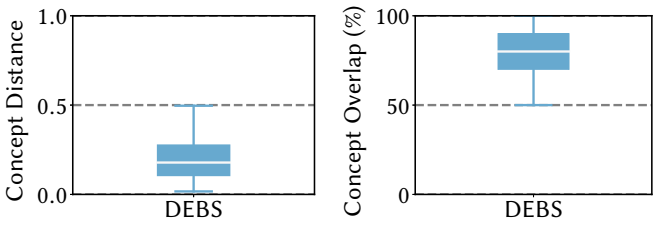
Considering the volatile nature of the input, the workload imposed on the SPE might become exorbitant and lead to a violation of the CQ's Service Level Objectives (SLOs). In our example, a temporal event might increase the *Volume* and the *Velocity* of the rides stream, and in turn delay the production of results. To address this problem, a lot of work has been done to enhance SPEs' adaptability. The most popular solution, involves elastically adding more resources to accommodate processing demands [21], [7], [22], [23], [24]. This solution has been shown to achieve the expected results in the long run, but causes significant performance overhead when used for short-lived input spikes.

A more suitable technique for short-term bursts is *load shedding*, which dictates dropping input tuples to produce results in a timely fashion at the expense of the results' accuracy [25]–[31]. This accuracy drop is in agreement with data analyst's needs, which often focus on the qualitative relationships among groups rather on the exact numerical result [32]. For example, in the query of Fig. 1 the analyst is mostly interested in the order of the top routes, rather than the average fare per route. The main challenge of load shedding is to downsample input without deteriorating the accuracy of a result.

Existing load shedding techniques require either (a) user input regarding individual tuples' utilities [25] or (b) historical data accumulated over time [26], [31], [33]. The first requirement is impractical for CQs that require extracting patterns

978-1-5386-5035-6/18/$31.00 ©2018 IEEE

Authorized licensed use limited to: University of Pittsburgh Library System. Downloaded on November 12,2025 at 15:55:56 UTC from IEEE Xplore. Restrictions apply.



a: Distance b: Overlap

Fig. 2: Real-world datasets feature varying *concept drift* characteristics, in terms of both inter-window concept distance and overlap.

in real time. For instance, in the query of Fig. 1 expecting the user to provide a tuple's utility based on the route is equivalent to requesting the top routes of each window. The second requirement is impractical due to the fact that real-world streams evolve over time.

A data stream's evolution over time is called *concept drift* and is a frequent phenomenon in the real world. As evidence, we produced the results of the example query presented in Fig. 1 on a real-world dataset. This dataset consists of taxi data from a New York City Taxi Company, which along with the CQ of Fig. 1 were part of the ACM DEBS 2015 Challenge [34]. We ran this query on the whole dataset and produced the result for each window. Fig. 2 illustrates the mean set difference (i.e., "concept distance") and mean set intersection (i.e., "concept overlap") among consecutive window results. It is apparent that the top routes (i.e., "concept") change among consecutive windows by up to 50%. This indicates that load shedding techniques, which rely on historical information to prioritize shedding, are destined to deteriorate. As a result, uniform load shedding is the only applicable technique in current SPEs, which is also stated in the work of Fu et al. [35].

Contributions: Motivated by the lack of a better solution for load shedding in modern SPEs that accounts for *concept drift*, we developed a new algorithm that can significantly improve the accuracy of load shedding. To reduce the runtime complexity of our proposed algorithm, we leverage the architecture of modern SPEs and present a design with almost zero overhead during processing. In detail, our contributions can be summarized as:

- Proposed a novel load shedding algorithm named **Concept-Driven** load shedding (CoD), which treats *concept drift* as a first-class characteristic. It is widely applicable to a variety of CQ types, and does not require user input for windows' *concepts*.
- Designed CoD for modern SPEs, so that no additional performance overhead is introduced.
- Performed experimental analysis on real data with varying *concept drift* characteristics to evaluate the performance of our proposed load shedding techniques in terms of error, data volume reduction, and runtime performance.

TABLE I: Symbol Index

Symbol	Overview
$\mathcal{Q}$	submitted CQ
S_i	input streams, $1 \leq i \leq N$
$e_1, \ldots, e_\infty$	tuples of S_i
$\mathcal{X}_i = (\tau, k, p)$	schema of S_i
$\mathcal{W}_\mathcal{Q} : S_i \rightarrow \{S_i^1, \ldots, S_i^\infty\}$	window function of $\mathcal{Q}$
R_w	result of window w (i.e., S_i^w)
C_w	concept of window w
$\hat{R}_w$	result of window w after load shedding
$\epsilon_\mathcal{Q}(R_w, \hat{R}_w)$	accuracy metric for $\mathcal{Q}$'s result

Outline: Section II presents the framework of our work. Section III introduces the concepts related to load shedding, followed by Section IV with existing algorithms. Next, in Section V we present our proposed algorithm, and in Section VI we present CoD's design on modern SPEs. Section VII presents our experimental results, followed by related work in Section VIII. Section IX concludes our work.

II. Prerequisites - Background

In this section we establish the framework on which our work is based, by describing the System, Query, and Data Models. Table I outlines the symbols we use in our formulation. Our work targets SPEs working both on multi-node (scale-out) and single-node (scale-up) environments. Examples of target SPEs are Storm [11], Heron [12], Apache Flink [15], and Spark Streaming [9].

A. Query Model

A user submits a CQ $\mathcal{Q}$, which is a transformation of input tuples to the output result. $\mathcal{Q}$ can contain one or more operations that can be either stateless (e.g., filter, map etc.) or stateful (e.g., window, aggregate, join etc.). The main difference among those two operator types is that the former produces zero or more output tuples for each input tuple, whereas the latter produces zero or more result(s) for a number of input tuples. The SPE turns $\mathcal{Q}$ into a logical execution plan, which is modeled by a directed acyclic graph (DAG). The DAG's vertices denote operations and the edges data transfers. Next, the SPE transforms the DAG to a physical execution plan, a process which entails instantiating execution workers for each operation (i.e., threads, processes etc.), creating the physical communication channels among different stages of execution (i.e., I/O buffers, network sockets etc.), initiating monitoring and reliability mechanisms, and commencing synchronization components. In this stage, tuple distribution operators are created, which transfer data among parallelized steps of the execution (see [36]). For simplicity, we will abstract $\mathcal{Q}$ as a single stateful operation. However, our model is able to accommodate queries with more than one stateful operations in its DAG.

The example query of Fig. 1 features a stateful operation, which is the group average per route, on a window of 15 minutes with a five minutes slide. In most SPEs, the windowing operation, the group aggregation, and the sort operation

Authorized licensed use limited to: University of Pittsburgh Library System. Downloaded on November 12,2025 at 15:55:56 UTC from IEEE Xplore. Restrictions apply.

are fused together, in order to avoid unnecessary network hops (Storm, Heron, and Flink do this). Depending on the parallelism degree defined for this query, the *limit* operation can be fused as well. If the aggregation takes place in parallel, then the limit needs to take place in a different operation.

B. Data Model

Data appear in the form of input streams S_i, where $1 \leq i \leq N$, and each S_i is a sequence of tuples $e_1, \ldots, e_\infty$ having a predefined schema $\mathcal{X}_i$. Depending on $\mathcal{Q}$, the schema can be represented as a triplet $\mathcal{X}_i = (\tau, k, p)$. In the event that window operations are defined in $\mathcal{Q}$, τ is (are) the attribute(s) used to assign the tuple to (a) window(s). In turn, a window can be time-, count-, or session-based and can either be tumbling or sliding. Windows can be modeled as functions of the form $\mathcal{W}_\mathcal{Q} : S_i \rightarrow \{S_i^1, \ldots, S_i^\infty\}$. The example query of Fig. 1 features a sliding time-based window: τ is the *time* field of each input tuple, the window's size is 15 minutes, and its slide is 5 minutes. This type of window is called *sliding* due to the fact that consecutive windows are overlapping. In a tumbling window, the size is equal to the slide. As far as k is concerned, it represents a set of attributes that work as an identifier for each tuple, given $\mathcal{Q}$ [36]. In the example query of Fig. 1, k is the *route* attribute of each input tuple and is used to group tuples together. Finally, p is the *payload* of a tuple and indicates attributes that are used neither in window assignment nor grouping. In the example of Fig. 1, *fare* is part of p, since it is used in the calculation of the aggregation's result. We have to mention that there are types of CQs, whose $k = \emptyset$. Those types can be scalar aggregations, or a combination of selection and/or projection queries.

C. Concept and Concept Drift

In the past, various definitions have been given to the *concept* of a stream. The most prevalent defines *concept* either as the underlying mechanism generating stream data, or the concept that is learned by applying $\mathcal{Q}$ to an input stream [37]. Naturally, *concept drift* describes the phenomenon when data are produced by non-stationary distributions [38] and evolve over time. Previous work on identifying *concept drift* in continuous streams, perceives the stream S_i as a continuous signal of information, without leveraging the windowing properties of $\mathcal{Q}$ [37], [39]. This constitutes the *concept drift* identification process difficult, because it requires the segmentation of S_i to "drift periods", which is a computationally heavy process.

We follow a more practical approach to *concept drift*, which leverages the windowing semantics of $\mathcal{Q}$, whose goal is to transform the input stream to the desired output. Therefore, an input stream S_i can be broken down to its window segments S_i^w and *concept drift* can be monitored on the window boundary. Thus, given a $\mathcal{Q}$ and its window definition $\mathcal{W}_\mathcal{Q}$, we define the *concept* of a window w as C_w (i.e., leverage the window boundary). As such, each window S_i^w will have its own concept C_w. The query shown in Fig. 1, exhibits concept drift of 30% on average on every window as shown in Fig. 2.

TABLE II: Examples of concept per query type

Operation	Concept	Utility for Load Shed
Aggregation	(μ, σ^2)	Sampling Rate (b)
Group Aggr.	Group Frequencies	Group Sampling Rate (b_g)
Equi-Join	Group Frequencies	Group Sampling Rate per stream

Concept drift can be quantified using two dimensions. First, *concept overlap* is the fraction of common elements among C_w and C_{w+1}. Second, *concept distance* is the similarity of C_w and C_{w+1} in terms of higher-level characteristics other than *concept overlap*.

Our query-driven definition of *concept* indicates that among different stateful operations, the notion of *concept* differs. Table II provides an overview of three of the most popular stateful operations in SPEs [1], our definition of *concept* C_w, and the utility of the former in load shedding. Starting from a mean-like aggregation (e.g., mean, count, variance etc.), C_w is determined by the measures of central tendency of the distribution followed by the participating attributes in the computation. Babcock et al. [26] has shown that when $\mathcal{Q}$ is a scalar aggregation, measuring the mean value (μ) and the variance (σ^2) of the distribution can provide useful insights for approximating its value: identify the proper sampling rate b for window S_i^w. Turning to grouped aggregations, like the one appearing in the example query of Fig. 1, C_w is the frequency of appearance of each group. This provides information about the distribution of groups in S_i^w and can be used to create a stratified sample for S_i^w when tuples need to be shed. This technique has been previously used in *Approximate Query Processing* for Data Warehouses [40], [41] so that an approximate value appears of each group. In order to build a representative stratified sample, the *concept* determines the sampling rate b_g for every group g in a window S_i^w. When $\mathcal{Q}$ is an equality join, the group frequency on each of the input streams can indicate the size of the result and the number of tuples per matched key [42]. Similar to a grouped aggregation, the *concept* is the frequency of each group appearing on each stream. This can be used during load shedding to identify the sampling rate of each group on each stream.

III. Load Shedding

In this section, we will present load shedding and formalize it as a minimization problem. A SPE enhanced with a load shedding mechanism comes with a load detection component. When the former detects that load exceeds the available processing capacity, it commences shedding tuples [3], [25], [26]. An end-to-end load shedding solution has to decide (a) **when**, (b) **where**, (c) **how much**, and (d) **what** to shed [25].

Decision (a) locates the point(s) in time most beneficial to commence load shedding, or can be a manual command given by the user. More often than not, this decision is independent with the rest of the operations and has to do with the monetary budget or available resources. Decision

[1]Those operations are the stateful operations offered by Storm, Heron, Flink, and Spark Streaming.

Authorized licensed use limited to: University of Pittsburgh Library System. Downloaded on November 12,2025 at 15:55:56 UTC from IEEE Xplore. Restrictions apply.

(b) finds the position in the execution DAG of $\mathcal{Q}$ to place shed operations. In previous work, load shedding took place in the source operators only [35]. This approach benefits from the fact that results produced are easier to justify, especially when they are enhanced with information from other sources. However, their accuracy is difficult to justify when sliding windows are used. For instance, in the query of Fig. 1 dropping a tuple might cause different windows to be produced. In turn, this will lead to missed window results. On the other hand, if load shedding takes place in a stateful operator of $\mathcal{Q}$, then results are produced based on the window boundaries of the CQ [28]. In our work, we follow this approach because we target applications that require production of results on every window. Finally, Decision (c), is related with the available processing capacity, the runtime complexity of $\mathcal{Q}$, and the input rate of source streams. In this paper, we consider this an expert's decision, and assume that the shedding rate is given to the SPE by the user. Currently, we are investigating dynamic methods for determining the shed rate in an online fashion.

Previously-proposed load shedding techniques aimed at maintaining results accuracy and focused on **what** to shed. *Semantic* load shedding prioritizes tuples for shedding [25]. Similarly, the system presented in [43] chooses a different shedding rate for classes of CQs with varying SLOs. When tuples are shed, the accuracy of the result is affected and there are various metrics for measuring error. For a given $\mathcal{Q}$ and a S_i^w, we will represent the result after processing all tuples as R_w. With load shedding in place, which we will represent as $\mathcal{S}$, a subset of tuples is processed. Therefore, an approximate answer for S_i^w is produced, which we will illustrate it as $\hat{R}_w$. Depending on the stateful operation of $\mathcal{Q}$, there are various metrics of quantifying $\hat{R}_w$'s error with respect to $\mathcal{Q}$. In our formulation, we will represent the error metric as $\epsilon_{\mathcal{Q}} : R_w, \hat{R}_w \rightarrow [0, 1]$. In essence, the lower the error, the more effective $\mathcal{S}$ is.

A. Focal point of this work: what to shed

In this work, we focus on **what** to shed and our goal is to improve load shedding's accuracy by prioritizing shedding of tuples based on their contents. In order to improve accuracy, a SPE needs to be aware of each window's *concept* C_w. Concretely, this is equivalent to knowing the utility QoS graph required by *Semantic* load shedding [25]. This way, load shedding would be prioritized based on tuples' importance according to C_w. We codify the problem of **what to shed** as the following minimization problem:

$$\begin{aligned} \underset{C_w}{\text{minimize}} \quad & \epsilon_{\mathcal{Q}}(R_w, \hat{R}_w) \\ \text{where} \quad & R_w = \mathcal{Q}(S_1^w, \ldots, S_l^w), \\ & \hat{R}_w = \mathcal{Q}(\mathcal{S}(S_1^w, \ldots, S_l^w, C_w)) \end{aligned} \tag{1}$$

In Equation 1, $\epsilon_{\mathcal{Q}}$ is the error metric for the result of $\mathcal{Q}$, R_w represents the result for S_i^w without shedding any tuples, and $\hat{R}_w$ represents the result for S_i^w when tuples are shed. $\mathcal{S}$ is the load shedding operation and C_w is the window's *concept*. In essence, Equation 1 states that **"what to shed"** is the search for C_w on every window, which can be a hard problem due to the uncharted future inputs and the existence of *concept drift* (Fig. 2).

Algorithm 1 Uniform Shedding

```
1: procedure UNIFORMSHED(S_i^w, b)
2:     T'_w ← Sample(S_i^w, b)
3:     return T'_w
```

IV. Existing Shedding Algorithms

Modern SPEs offer strict delivery semantics, which lead to applying the processing logic into ordered windows [9], [12], [13], [15], [44]. As mentioned in Section II, tuples are assigned to windows according to $\mathcal{W}_{\mathcal{Q}}$, and the processing logic of $\mathcal{Q}$ is applied to a complete window. For simplicity, we present load shedding algorithms on complete windows (similar to the approach of [28]) instead of single tuples at a time (in Section VI we show the internal operations of forming a window in a SPE).

A. Normal Execution (Baseline)

In order to approach the minimization problem of Eq. 1, we establish normal execution as our *Baseline*. This would represent normal execution on S_i^w without shedding any tuples (i.e., $\mathcal{S} : S_i^w \rightarrow S_i^w$ in Eq. 1). The *Baseline* features zero shedding runtime overhead since the whole window is processed, and maximum accuracy (i.e., $\epsilon_{\mathcal{Q}} = 0$). With *Baseline* there is no need to estimate C_w and its drawback is that the data volume sent for processing will be the whole window.

B. Uniform Shedding (State of the Art)

As discussed in Sections I and III, when C_w is unavailable and input streams(s) feature *concept drift*, only *Uniform* load shedding can be applied. Recent work presented in [35] explains that *Uniform* load shedding is used in industrial setups. Therefore, in this work we consider *Uniform* load shedding as the state of the art.

Uniform load shedding materializes by extracting a uniform sample from S_i^w and sending it for processing. Algorithm 1 presents an outline of *Uniform* load shedding which receives two arguments: S_i^w and a shed parameter b. The latter can either represent a shed bias (i.e., probability) or a percentage of S_i^w that will be processed. In Algorithm 1, a call to method *Sample* (line 2) creates a uniform sample (e.g., binomial, reservoir etc.).

Uniform load shedding makes no effort in estimating C_w. As a result, the error $\epsilon_{\mathcal{Q}}$ is sensitive to the sampling rate b. *Uniform*'s runtime cost is $O(c|S_i^w|)$, where c is the cost of the random process. Finally, the data volume sent for processing is proportional to the bias parameter b: *the higher the bias, the higher the processing cost*. *Uniform* load shedding presents an error-agnostic approach and focuses solely on reducing the processing load imposed to the SPE.

Authorized licensed use limited to: University of Pittsburgh Library System. Downloaded on November 12,2025 at 15:55:56 UTC from IEEE Xplore. Restrictions apply.

Algorithm 2 Concept-driven load shedding

```
1: procedure CODSHED(S_i^w, b, Q)
2:     T_w ← Sample(S_i^w, b)
3:     C_w ← Q(T_w)
4:     T'_w ← ∅
5:     for all t ∈ S_i^w do
6:         if ¬Shed(t, C_w) then
7:             T'_w ← T'_w + t
8:     return T'_w
```

V. Concept-Driven Load Shedding Algorithm

In order to improve accuracy, a SPE needs to be able to estimate C_w on every window w. Our proposed load shedding technique is named **Co**ncept-**D**riven (CoD) and estimates C_w by applying $\mathcal{Q}$ on a uniform sample of S_i^w. Concretely, CoD decouples sampling from shedding to get insights for improving accuracy. CoD is motivated by work done on approximate query processing, such as Aqua [45] and BlinkDB [40]. Those run queries on samples extracted from the dataset and provide approximate answers. In contrast, CoD uses a sample to estimate C_w, which in turn is utilized for shedding tuples.

Algorithm 2 outlines CoD, whose input consists of a window S_i^w, a load shedding percentage b, and $\mathcal{Q}$. b controls the size of uniform sample T_w, to which $\mathcal{Q}$ will be executed for retrieving C_w. Alg. 2's output consists of $T'_w \subset S_i^w$, which is forwarded for processing. In Alg. 2, $\mathcal{Q}$ is required so that an estimation of the window's *concept* is extracted (see Sec. II-C): Depending on the query type, Table II presents the information extracted to improve result's accuracy. For example, in the query of Fig. 1, C_w will be the frequency of each group *route* appearing in S_i^w. Then, depending on b, each *route* would be assigned a portion of b (i.e., b_r). In essence, $C_w = \{b_1, \ldots, b_r\}$ is the sampling rate for each *route*. By the time C_w is established, the tuples of S_i^w are scanned once more, and each tuple t is passed to a $Shed$ method along with C_w (line 6). $Shed$ is responsible for deciding whether to shed a tuple, by examining whether tuple t's extracted attributes are part of C_w. If not, the tuple is shed; otherwise it is included in T_w.

CoD requires two passes over S_i^w and has a runtime complexity of $O(c|S_i^w| + b|S_i^w| + c|S_i^w|)$. The first operand denotes the cost of creating the sample, the second operand is the cost of executing $\mathcal{Q}$ on a sample of size $b|S_i^w|$, and the third operand is the cost of applying load shedding to S_i^w. In essence, CoD's runtime cost is two times the runtime cost of *Uniform* (since the dominant factor is the scan of the window and not applying $\mathcal{Q}$ on the sample). This can be a very high cost and in the next Section we present an efficient design that turn's CoD runtime cost equal to that of *Uniform*'s.

VI. Designing CoD to eliminate overhead

The design of modern SPEs provides an opportunity to avoid incurring additional overhead when CoD is applied.

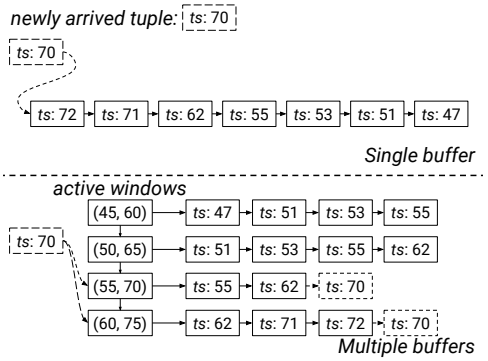


Fig. 3: Tuple arrival: *Single* and *Multiple* Buffer(s) design operations.

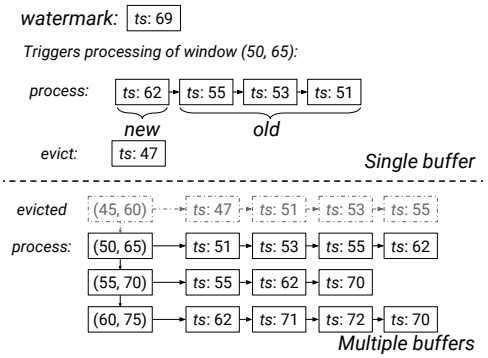


Fig. 4: Watermark arrival: *Single* and *Multiple* Buffer(s) design operations.

In this section, we present the details of event time window processing in modern SPEs, CoD's implementation, and CoD's implementation details for stateful queries.

A. Event time processing on modern SPEs

When a tuple arrives at a stateful operator, it is stored in the operator's internal buffer(s). Fig. 3 illustrates the two widely used designs among existing SPEs: (a) *Single Buffer* dictates that all tuples are stored in a single buffer based on their order of appearance. This design is followed by Storm and Heron and its advantage is that each tuple is stored only once. (b) *Multiple Buffer* design dictates that a copy of the input tuple is stored to all the buffers of the corresponding windows. Flink uses the Multiple Buffer design and its merit is that when the time comes for processing, the window is ready. In the example of Fig. 3, a tuple with timestamp 70 arrives and participates in windows: $(55, 70)$, $(60, 75)$, $(65, 80)$, and $(70, 85)$. With the *Single Buffer* design, the tuple is appended to the buffer and the operator waits until the next one arrives. In contrast, with the *Multiple Buffer* design, the tuple's timestamp is extracted, and the windows that it belongs to are determined (the last window that the tuple participates in is given by $\tau_l = \tau - ((\tau + s) \mod s)$, where s is $\mathcal{W}_\mathcal{Q}$'s slide, and the rest are found iteratively). A copy of the tuple is stored in each buffer for each window.

As previously shown in the Dataflow system [44], SPEs which support event time window processing, make use of

Authorized licensed use limited to: University of Pittsburgh Library System. Downloaded on November 12,2025 at 15:55:56 UTC from IEEE Xplore. Restrictions apply.

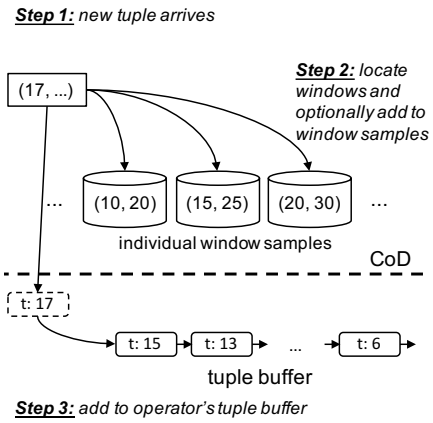


Fig. 5: CoD tuple arrival: the tuple is optionally added to the windows it belongs to, and also stored in the buffer space.

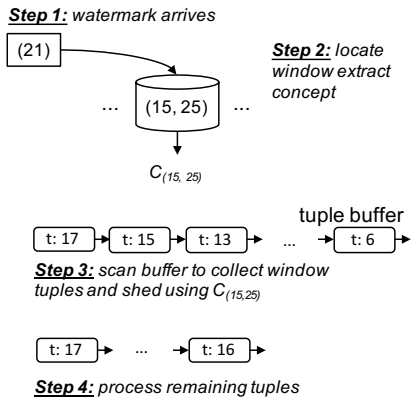


Fig. 6: CoD watermark arrival: the concept is extracted from the sample and then tuples are shed based on it.

periodic watermarks. Those are tuples having only a timestamp (τ_W), and indicate that all tuples with $\tau \leq \tau_W$ have been observed by the system. A watermark is akin to a contract that processing will have access to all tuples up to a particular point in time, and that no late tuples will be encountered. In addition, the watermark mechanism is used as a checkpointing mechanism to guarantee exactly-once processing semantics [17]. Watermarks are produced by data source operators in a periodic fashion. Every time a window operator receives a watermark, it triggers the processing of all affected windows.

At the arrival of the watermark, the window operator has to prepare all non-processed windows ending on (or before) τ_W. The operator extracts the tuples for each window, and stages them for processing. With *Single Buffer* design, the operator scans the buffer and gathers all the tuples belonging to a particular window. At the same time, it evicts expired tuples (those that they carry a timestamp earlier than the start of the current window). Optionally, the window operator might perform an additional scan on the window to separate tuples processed for the first time. This scan is done on SPEs that offer incremental processing (e.g., Storm, Heron). Turning to the *Multiple Buffers* design, the operator picks the appropriate window, and it passes it the processing logic. Fig. 4 presents the window preparation process when the watermark with timestamp 69 arrives. With a *Single Buffer* design, the operator scans its buffer and collects the tuples of window $(50, 65)$, and marks tuple 47 for eviction. Finally, the operator sends tuples for processing. With a *Multiple Buffer* design, the operator simply picks the buffer for the corresponding window and sends the tuples for processing.

B. CoD implementation

In both designs, tuple storage is decoupled from window processing. Concretely, with the watermark-based execution, processing is postponed until the watermark has been received. This way, CoD can be designed to avoid additional scans and maintain a runtime cost similar to that of *Uniform*.

In order for CoD to avoid the additional scan of a window, uniform sampling is performed incrementally. When a tuple arrives, the operator determines the windows that the tuple belongs to. Also, for each window, CoD maintains a uniform sample of size b. The incoming tuple is added on each window's sample based on a random process (i.e., coin flip). Fig. 5 illustrates the steps that are followed by CoD when a new tuple arrives. **Step 1:** the tuple's timestamp is extracted and it is used to identify the windows that the tuple participates. **Step 2:** for each of those windows, a random process (e.g., a coin-flip) determines whether the tuple will be included in the sample. In our implementation, each sample is a reservoir sample, in order to maintain samples of size b for each window. **Step 3:** the tuple is stored to the operator's storage space.

At a watermark arrival, CoD uses the uniform samples to extract window's *concept* C_w. **Step 1:** Extract watermark's timestamp and identify window to be processed. **Step 2:** CoD extracts C_w from the corresponding sample. **Step 3:** in a single scan of the tuple buffer CoD locates the tuples for the window, uses C_w to prioritize tuples for shedding, and evicts expired ones. **Step 4:** surviving tuples are sent for processing and the sample for the window is discarded.

As we presented in Sec. IV-B, *Uniform*'s runtime cost requires one scan over S_i^w. This action can be performed when the window is scanned to extract the window's tuples. CoD's design results in a similar runtime cost, with a time complexity of $O(c|S_i^w|)$ (since the dominant factor is the scan of the window). This cost is equivalent to *Uniform* load shedding runtime cost.

C. Implementation for different query types

As has been illustrated in Table II, different stateful operations have a different *concept*. In this section we provide details of CoD's sampling process for different query types.

1) Scalar User-Defined Aggregations: When $\mathcal{Q}$ is a scalar User-Defined Aggregation (UDA) then CoD will maintain a uniform sample for each window. For instance, if $\mathcal{Q}$ is the arithmetic mean of tuples' payloads (p), then during tuple arrival, based on a random process tuple's p will be included

Authorized licensed use limited to: University of Pittsburgh Library System. Downloaded on November 12,2025 at 15:55:56 UTC from IEEE Xplore. Restrictions apply.

TABLE III: Dataset Characteristics

Dataset	Size	Queries	Mean Window Size (tuples)
DEBS	32 GB	Group Aggr.	$\sim$ 100K
GCM	16 GB	Group Aggr.	320K
DEC	175 MB	Aggr.	46K

in the sample. When a watermark arrives, CoD will extract the mean value μ and the variance σ^2 from the window sample (see Table II). With this information, it can decide the sampling rate required to provide an answer based on a user's accuracy requirements [26].

2) User-Defined Aggregations: When $\mathcal{Q}$ is a grouped UDA, then during tuple arrival CoD maintains the frequency of appearance of each group. Given a shed bias b and a window's size, CoD can create a stratified sample with a proportional representation of each group. Concretely, for each group g, CoD has to determine its sampling rate b_g. As indicated in [40], [41], [45], this technique will ensure that every group g will appear in the result. When a watermark arrives, CoD extracts each tuple's k attribute(s), and based on the tuple's group, it maintains a sample of size $b_g|S_i^w|$. In comparison with *Uniform*, this stratified sampling approach is expected to lead to more accurate results and zero missing groups.

3) Two-way Equality Joins: If $\mathcal{Q}$ is an equality join between two streams, then CoD needs to maintain a histogram of groups for each of the two streams [42]. This way, CoD will be able to know which groups have a higher likelihood of forming a result. When a watermark arrives, CoD identifies the intersection of the two histograms: for each group g that appears in both histograms, a sample rate $b_g^{\{1,2\}}$ is determined based on the overall shed bias b, for each stream. Similar to grouped UDAs, when the window is scanned, a stratified sample for each stream and each group on each stream is maintained and pushed for processing.

VII. Experimental Evaluation

In our experimental evaluation, our goal is to measure performance, error, and the ability to decrease data volume in real world conditions. We implemented *Uniform* and CoD algorithms on Apache Storm v1.2, by extending the `WindowManager` class. For *Uniform*, our `ShedWindowManager` class requires the shed percentage parameter b. When a watermark arrives, a binomial process ($p = b$) determines whether the tuple is shed or not. Even if a tuple is shed, it is maintained in order to acknowledge its acceptance to the upstream operator. For CoD, our `ConceptShedWindowManager` receives the shed percentage parameter b, which controls the size of the sample for each window. Every time a tuple arrives, CoD augments the corresponding window samples by using a similar binomial process as in *Uniform*. At watermark arrival, the concept is extracted from the corresponding sample, and tuples are shed accordingly.

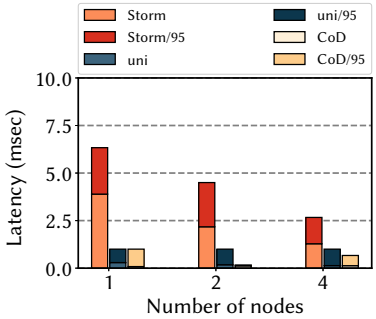


Fig. 7: Scalability of normal execution compared to *Uniform* load shedding and CoD, with the shedding ratio set to 98%.

A. Experimental Setup

We conducted our experiments on a cluster consisting of 5 AWS r4.xlarge nodes. Each node ran on Ubuntu Linux 16.04 and OpenJDK v1.8. Each node had access to 4 virtual CPUs of an Intel Xeon E5-2686 v4 and 32GBs of RAM. One node was set up as the master. It had a long-running single-instance ZooKeeper server (v3.4.10) and an Apache Storm Nimbus process. The rest of the nodes ran a single Storm Supervisor process and were configured with up to 3 execution threads (one was reserved for Storm's acknowledgment service). In all our experiments, we enabled Storm's acknowledgment mechanism to guarantee processing of all tuples. All our experiments were repeated five times and the numbers reported are the averages of all runs.

B. Datasets

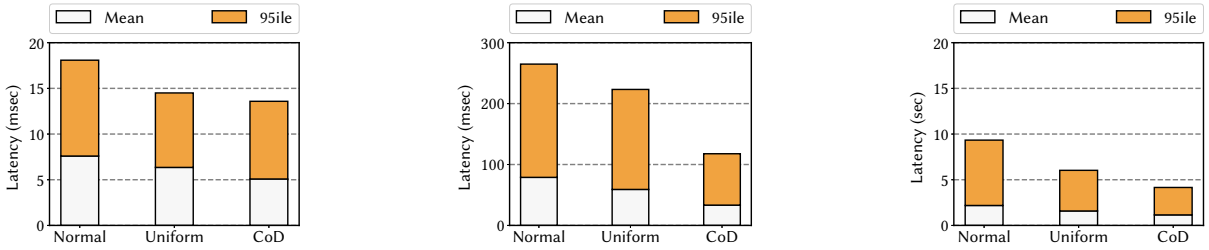
We used three real-world datasets, with varying sizes, data characteristics, and window queries. Two of the datasets featured grouped aggregations and one scalar aggregation. Table III presents for each dataset its total size, the aggregation type, and the average window size in tuples. We provide additional details for each dataset below:

ACM DEBS Challenge Dataset (DEBS): This dataset consists of 32 GB of data accumulated from a New York Taxi company over a month. In its original form, this dataset comes with two sliding window queries [34]. We calculated the mean fare per route for sliding windows of 30 minutes, with a 10 minute slide. We used a single source operator, a varying number of window aggregation operators, and a single collector bolt that accumulates routes the aggregation result.

Google Cluster Monitoring Dataset (GCM): This dataset contains information about jobs submitted in one of Google's clusters [46]. From this dataset, we used part of the *task-events* table and ran a sliding window grouped aggregation, taken from [16]. This query calculates the average CPU time requested by each scheduling class, on a 60 minute window with a 30 minute slide. Compared to DEBS this dataset features less groups and smaller window sizes.

DEC Network Monitoring Dataset (DEC): This dataset is the smallest in size that we used in our analysis. Previously, it has been used to measure *Uniform* load shedding's accuracy [26] and it consists of network packets monitored over an

Authorized licensed use limited to: University of Pittsburgh Library System. Downloaded on November 12,2025 at 15:55:56 UTC from IEEE Xplore. Restrictions apply.



a: DEC b: GCM c: DEBS

Fig. 8: Overall average and 95 percentile window latency with 4 worker nodes.

hour. The query that we used for this dataset is the calculation of the average packet size on sliding windows of size 45 seconds and slide of 15 seconds.

C. Experimental Results

Below, we present our experimental results on scalability (Section VII-C1), performance (Section VII-C1), accuracy (Section VII-C3), and data volume reduction (Section VII-C4). To this end, we compared normal execution with the current state-of-the-art in load shedding (i.e., *Uniform*) and CoD.

1) Scalability (Fig. 7): First, we examined the scalability of each approach by measuring the average and 95 percentile window processing latency on the DEC dataset. The reason we picked DEC for this experiment is because it is a processing-heavy aggregation, without any need for memory. Therefore, this dataset poses a favorable use-case for normal execution, since the workload is not memory-heavy. Also, we set the shedding rate (b) to 98% for both *Uniform* and CoD, since our analysis has shown that it leaves enough data to produce results with less than 10% error (see Section VII-C3). In addition, we used the whole dataset and we doubled the number of server nodes (i.e., worker nodes) that participate in the process. Fig. 7 presents the average and 95 percentile latency for normal (labeled as "Storm"), *Uniform* load shedding (labeled as "uni"), and CoD load shedding (labeled as "CoD").

As can be seen, Storm has the highest average and 95 percentile latency. The reason for this is due to the fact that processing takes place in the whole window and all tuples are scanned. *Uniform*'s performance is faster due to the fact that a fraction of the window is processed. The same is the case with CoD. Both settings with shedding achieve significantly better average processing latency compared to normal and more than four times better 95 percentile latency. Another important observation is that with load shedding, better processing latency is achieved with a fraction of the resources: CoD and *Uniform* running with one node achieve better latency compared to normal execution running on four nodes.

Take-away: With CoD, processing takes place on a fraction of the window, and one requires much less resources to achieve better processing latency compared to Storm without any load shedding.

2) Performance (Fig. 8): Next, we measured overall mean and 95-percentile latency for all three datasets. Compared to the scalability experiment (Sec. VII-C2) we measure end-to-end latency on the stateful operator, from the watermark arrival until the window result is pushed to the next operator. For this experiment, we set the number of worker nodes to four (i.e., utilize all our resources), and the shed-bias to 97% (i.e., less than three percent of the window gets processed). For all datasets, we used a single source operator, and a sink operator.

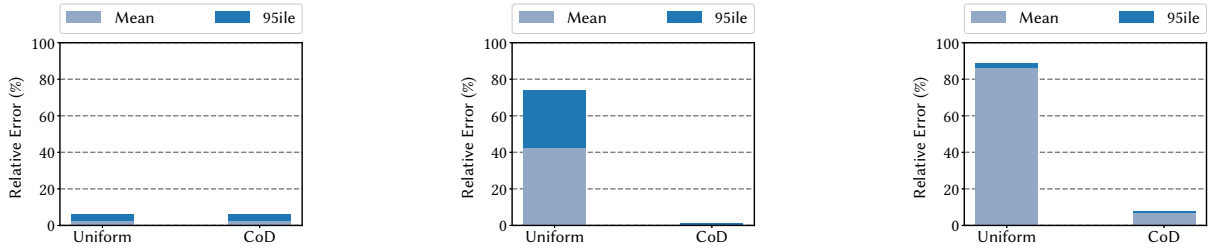
Fig. 8a presents the overall latency for the DEC dataset. As can be seen, the overall latency is lower for *Uniform* and CoD. For DEC, most of the time is spent on window preparation (i.e., window scan). Therefore, the performance gap between normal and the load shedding methods is small. However, this is not the case in GCM and DEBS. Fig. 8b illustrates that CoD and *Uniform* present better performance compared to normal execution. The reason is that GCM requires a grouped aggregation, which can introduce additional load to each worker. Fig. 8c presents a similar picture. CoD manages to produce a result faster than normal Storm, and up to 2.25 times faster for the 95 percentile case. In our prototype implementation we measured that *Uniform* was slower compared to CoD due to the frequency of the binomial process (i.e., on every tuple of the window). This is not the case with CoD, since tuples of infrequent groups are simply added to the window.

Take-away: CoD improves overall latency and it can produce a result up to 2.25 times faster compared to normal Storm.

3) Accuracy (Fig. 9): For the two load shedding techniques, we measured the average error and the 95-percentile error among all windows on all datasets. For the error metric we used the relative error for the aggregation result of DEC, and for DEBS and GCM the average error among all groups for a given window. In addition, for each dataset we picked a different shed bias (b) which was 99%, 98%, and 98% for DEC, GCM, and DEBS. In the case of CoD, b works as the size of the sample created to identify C_w.

Fig. 9a shows that both *Uniform* and CoD produce comparable results for DEC. This happens because DEC features a scalar aggregation query. Therefore, CoD creates a uniform sample, which is equivalent to *Uniform* and the resulting accuracy is the same for both techniques. Fig. 9b illustrates the error for GCM, and the difference between the two becomes significant. In detail, *Uniform* sheds infrequent groups from the result, which in turn make the average error more than 40%. This is not the case with CoD, which employs stratified sampling for each group, by measuring their frequency. Then,

Authorized licensed use limited to: University of Pittsburgh Library System. Downloaded on November 12,2025 at 15:55:56 UTC from IEEE Xplore. Restrictions apply.



a: DEC b: GCM c: DEBS

Fig. 9: Average and 95 percentile relative error for DEC, GCM, and DEBS.

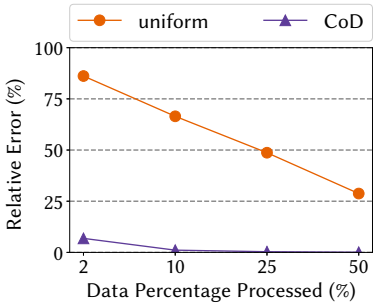


Fig. 10: CoD achieves much better error with a fraction of the data required by *Uniform*.

CoD determines a shed bias for each group, and when it scans the window creates a sample for each group. As a result, no groups are missed from the final result and the error remains below 10%. A similar behavior can be seen with DEBS, whose results are shown in Fig. 9c. Due to the fact that DEBS features more groups, *Uniform*'s error is more than 80%. This is not the case with CoD which manages to maintain both average and 95-percentile error close to 10%.

Take-away: CoD is able to identify the *concept* of each window and maintain accuracy more than 90%. Compared to *Uniform*, CoD can achieve more than an order of magnitude better accuracy.

4) Data Volume Reduction (Fig. 10): In the last experiment we wanted to analyze the drop of value in error while the data volume processed increases. To this end, we utilized all the workers in our infrastructure and executed the DEBS workload with varying b (shed bias) values: 98%, 90%, 75%, and 50%. On each run, we measured the average relative error achieved by each load shedding technique: *Uniform* and CoD.

Fig. 10 illustrates the average relative error when a different percentage of the window is processed. As far as *Uniform* is concerned, when the data percentage processed increases, the average relative error drops (almost) linearly. After examining the quality of the results, *Uniform* achieves big values for errors for two reasons. First, groups that do not appear often in the window are completely dropped, which leads to 100% error for them. Second, groups that appear frequently have some of their values dropped, which leads to high error values for those as well. On the other hand, CoD identifies the right amount of sampling needed per group by employing stratified sampling. Therefore, each group is represented proportionally in the final result, and the average relative error remains significantly low. In fact, for a data percentage value 25 times smaller, CoD achieves much higher accuracy compared to *Uniform*.

Take-away: CoD makes it feasible to process significantly less data while maintaining a very high accuracy.

VIII. Related Work

Throughout this paper we presented previous work done in load shedding [25], [26], [28], [31], [33], [47]. In addition, *Uniform* load shedding has appeared in a distributed version [30]. A control-based approach to load shedding is presented in [29] and the work of Gedik et al. [27] focuses on load shedding for join operations. Unfortunately, none of those feature a general mechanism for estimating C_w, and in turn they are not robust against *concept drift*. The work presented in [33] presents two heuristics for estimating the *concept* for equality joins. However, those algorithms are targeted for joins and rely on frequency-based estimation, which might not efficiently estimate the *concept* on all CQ types (i.e., grouped median). In addition, those heuristics do not present a systematic approach for detecting *concept drift*. Furthermore, prior work focused on optimization of continuous queries under resource constraints [27], [31], [47]. [47] presented an optimization framework based on the resource constraints of window joins on unbounded streams. Moreover, work on load shedding has looked at other dimensions beyond correctness. The work of Kalyvianaki et al. [48] presents an algorithm that focuses on fair load shedding among federated SPEs. The work of Pham et al. [43] presents load shedding in agreement with QoS of CQs with different priority classes.

Finally, a lot of work has been done on approximation data structures for stream processing [49]–[53]. However, those approaches aim at particular classes of CQs and cannot be used for multiple types of operators. Finally, sampling-based approximation processing has been previously proposed for relational databases [40], [45]. In the case of data streams, data arrive continuously and the their characteristics are unavailable for the SPE to analyze and pick an optimal sampling strategy.

IX. Conclusions

In this paper we revisited load shedding in modern SPEs. Our investigation on load shedding has revealed two shortcomings of previously proposed techniques. Namely, that

Authorized licensed use limited to: University of Pittsburgh Library System. Downloaded on November 12,2025 at 15:55:56 UTC from IEEE Xplore. Restrictions apply.

previously proposed techniques are (a) brittle against the inherent *concept drift* of data streams, and (b) rely on knowing tuples' importance at query submission. These shortcomings constitute *Uniform* load shedding the only applicable solution for modern SPEs. To improve the accuracy of the current state of the art (i.e., *Uniform*), we proposed CoD, a novel technique for load shedding, which relies on estimating a window's *concept* before shedding tuples. Even though CoD requires two scans of a window, we describe a design that incurs zero overhead for modern SPEs by taking advantage of their architecture. We implemented CoD for Apache Storm and examined its performance, accuracy, and data volume reduction capabilities compared to normal execution and *Uniform* load shedding. Our experiments with real workloads show that CoD manages to (a) reduce processing costs by up to 2.25x compared to normal execution, (b) achieve more than an order of magnitude better accuracy compared to the state of the art in load shedding, and (c) reduce the data volume significantly.

ACKNOWLEDGMENT

The authors would like to thank the reviewers for their useful feedback. This research was partially supported by NSF Award CNS-1739413.

REFERENCES

[1] J. Chen, D. J. DeWitt, F. Tian *et al.*, "Niagaracq: A scalable continuous query system for internet databases," in *SIGMOD*, 2000, pp. 379–390.
[2] S. Babu and J. Widom, "Continuous queries over data streams," *SIGMOD Record*, vol. 30, no. 3, pp. 109–120, 2001.
[3] D. J. Abadi, D. Carney *et al.*, "Aurora: A new model and architecture for data stream management," *VLDBJ*, vol. 12, no. 2, pp. 120–139, 2003.
[4] S. Chandrasekaran, O. Cooper, A. Deshpande *et al.*, "Telegraphcq: Continuous dataflow processing for an uncertain world," in *CIDR*, 2003.
[5] H. Andrade, B. Gedik *et al.*, "Processing high data rate streams in system s," *J. Parallel Distrib. Comput.*, vol. 71, no. 2, pp. 145–156, 2011.
[6] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel *et al.*, "The design of the Borealis stream processing engine," in *CIDR*, 2005.
[7] V. Gulisano *et al.*, "Streamcloud: An elastic and scalable data streaming system," *TPDS*, vol. 23, no. 12, pp. 2351–2365, 2012.
[8] R. Ananthanarayanan *et al.*, "Photon: Fault-tolerant and scalable joining of continuous data streams," in *SIGMOD*, 2013, pp. 577–588.
[9] M. Zaharia *et al.*, "Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters," in *HotCloud*, 2012.
[10] D. G. Murray, F. McSherry, R. Isaacs *et al.*, "Naiad: A timely dataflow system," in *SOSP*, 2013, pp. 439–455.
[11] A. Toshniwal, S. Taneja, A. Shukla *et al.*, "Storm@twitter," in *SIGMOD*, 2014, pp. 147–156.
[12] S. Kulkarni, N. Bhagat, M. Fu *et al.*, "Twitter heron: Stream processing at scale," in *SIGMOD*, 2015, pp. 239–250.
[13] B. Chandramouli, J. Goldstein, M. Barnett *et al.*, "Trill: A high-performance incre- mental query processor for diverse analytics," in *PVLDB*, 2015, pp. 401–412.
[14] Y. Wu and K. L. Tan, "Chronostream: Elastic stateful stream computation in the cloud," in *ICDE*, 2015, pp. 723–734.
[15] P. Carbone *et al.*, "Apache flink™: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, 2015.
[16] A. Koliousis, M. Weidlich, R. Castro Fernandez *et al.*, "Saber: Window-based hybrid stream processing for heterogeneous architectures," in *SIGMOD*, 2016, pp. 555–569.
[17] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in apache flink®: Consistent stateful distributed stream processing," in *PVLDB*, 2017, pp. 1718–1729.
[18] Amazon, "Amazon kinesis," https://aws.amazon.com/kinesis, 2018.
[19] Microsoft, "Microsoft azure streams," https://azure.microsoft.com/en-us/services/stream-analytics/, 2018.
[20] Google, "Google cloud dataflow," https://cloud.google.com/dataflow/.
[21] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K. Wu, "Elastic scaling of data parallel operators in stream processing," in *IEEE IPDPS*, 2009, pp. 1–12.
[22] R. Castro Fernandez *et al.*, "Integrating scale out and fault tolerance in stream processing using operator state management," in *SIGMOD*, 2013.
[23] N. R. Katsipoulakis, C. Thoma, E. A. Gratta *et al.*, "Ce-storm: Confidential elastic processing of data streams," in *SIGMOD*, 2015, pp. 859–864.
[24] T. N. Pham, N. R. Katsipoulakis, P. K. Chrysanthis, and A. Labrinidis, "Uninterruptible migration of continuous queries without operator state migration," *SIGMOD Record*, vol. 46, no. 3, pp. 17–22, 2017.
[25] N. Tatbul, c. Uğur, S. Zdonik, C. M., and M. Stonebraker, "Load shedding in a data stream manager," in *PVLDB*, 2003, pp. 309–320.
[26] B. Babcock, M. Datar, and R. Motwani, "Load shedding for aggregation queries over data streams," in *ICDE*, 2004, pp. 350–.
[27] B. Gedik, K. L. Wu, P. S. Yu, and L. Liu, "Adaptive load shedding for windowed stream joins," in *CIKM*, 2005.
[28] N. Tatbul and S. Zdonik, "Window-aware load shedding for aggregation queries over data streams," in *PVLDB*, 2006.
[29] Y. Tu, S. Liu, S. Prabhakar, and B. Yao, "Load shedding in stream databases: A control-based approach," in *PVLDB*, 2006.
[30] N. Tatbul, U. Çetintemel, and S. Zdonik, "Staying fit: Efficient load shedding techniques for distributed stream processing," in *PVLDB*, 2007.
[31] U. Srivastava and J. Widom, "Memory-limited execution of windowed stream joins," in *PVLDB*, 2004.
[32] A. Parameswaran, "Visual data exploration: A fertile ground for data management research," http://wp.sigmod.org/?p=2342, 2018.
[33] A. Das, J. Gehrke, and M. Riedewald, "Approximate join processing over data streams," in *SIGMOD*, 2003, pp. 40–51.
[34] Z. Jerzak and H. Ziekow, "The debs 2015 grand challenge," in *DEBS*, 2015, pp. 266–268.
[35] M. Fu, S. Mittal, V. Kedigehalli, K. Ramasamy *et al.*, "Streaming@twitter," *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 15–27, 2015.
[36] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis, "A holistic view of stream partitioning costs," *PVLDB*, vol. 10, no. 11, pp. 1286–1297, 2017.
[37] H. Wang, W. Fan, P. S. Yu, and J. Han, "Mining concept-drifting data streams using ensemble classifiers," in *KDD*, 2003, pp. 226–235.
[38] R. Klinkenberg and T. Joachims, "Detecting concept drift with support vector machines," in *ICML*, 2000.
[39] J. Gama *et al.*, "A survey on concept drift adaptation," *ACM Comput. Surv.*, vol. 46, no. 4, pp. 44:1–44:37, 2014.
[40] S. Agarwal *et al.*, "Blinkdb: queries with bounded errors and bounded response times on very large data," in *Eurosys*, 2013, pp. 29–42.
[41] Y. Park, B. Mozafari, J. Sorenson, and J. Wang, "Verdictdb: Universalizing approximate query processing," in *SIGMOD*, 2018, pp. 1461–1476.
[42] S. Acharya, P. B. Gibbons *et al.*, "Join synopses for approximate query answering," in *ACM SIGMOD*, 1999, pp. 275–286.
[43] T. Pham, P. Chrysanthis, and A. Labrinidis, "Avoiding class warfare: managing continuous queries with differentiated classes of service," *VLDBJ*, vol. 25, no. 2, pp. 197–221, 2016.
[44] T. Akidau *et al.*, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," in *PVLDB*, 2015, pp. 1792–1803.
[45] S. Acharya, P. Gibbons *et al.*, "Aqua: A fast decision support systems using approximate query answers," in *PVLDB*, 1999, pp. 754–757.
[46] G. Corp., "Cluster monitoring dataset, https://github.com/google/cluster-data." [Online]. Available: https://github.com/google/cluster-data
[47] J. Kang, J. F. Naughton, and S. D. Viglas, "Evaluating window joins over unbounded streams," in *ICDE*, 2003, pp. 341–352.
[48] E. Kalyvianaki, M. Fiscato, T. Salonidis, and P. Pietzuch, "Themis: Fairness in federated stream processing under overload," in *SIGMOD*, 2016, pp. 541–553.
[49] P. Flajolet *et al.*, "Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm," in *AOFA*, 2007.
[50] S. Heule *et al.*, "Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm," in *EDBT*, 2013, pp. 683–692.
[51] F. Bin, D. Andersen, M. Kaminsky, and M. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *CoNEXT*, 2014, pp. 75–88.
[52] S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *PVLDB*, 2002, pp. 346–357.
[53] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri, "Medians and beyond: new aggregation techniques for sensor networks," in *SenSys*, 2004, pp. 239–249.

Authorized licensed use limited to: University of Pittsburgh Library System. Downloaded on November 12,2025 at 15:55:56 UTC from IEEE Xplore. Restrictions apply.