In the Land of Data Streams where Synopses are Missing, One Framework to Bring Them All

Rudi Poepsel-Lemaitre*, Martin Kiefer*, Joscha von Hein*,
Jorge-Arnulfo Quiané-Ruiz*,∓, Volker Markl*,∓

*Technische Universität Berlin (TU Berlin) ∓German Research Center for Artificial Intelligence (DFKI)
[r.poepsellemaitre,martin.kiefer,j.vonhein,jorge.quiane,volker.markl]@tu-berlin.de

ABSTRACT

In pursuit of real-time data analysis, approximate summarization structures, i.e., synopses, have gained importance over the years. However, existing stream processing systems, such as Flink, Spark, and Storm, do not support synopses as first class citizens, i.e., as pipeline operators. Synopses' implementation is upon users. This is mainly because of the diversity of synopses, which makes a unified implementation difficult. We present Condor, a framework that supports synopses as first class citizens. Condor facilitates the specification and processing of synopsis-based streaming jobs while hiding all internal processing details. Condor's key component is its model that represents synopses as a particular case of windowed aggregate functions. An inherent divide and conquer strategy allows Condor to efficiently distribute the computation, allowing for high-performance and linear scalability. Our evaluation shows that Condor outperforms existing approaches by up to a factor of 75x and that it scales linearly with the number of cores.

PVLDB Reference Format:
Rudi Poepsel-Lemaitre, Martin Kiefer, Joscha von Hein, Jorge-Arnulfo Quiané-Ruiz, Volker Markl. In the Land of Data Streams where Synopses are Missing, the One Framework to Bring Them All. PVLDB, 14(10): 1818 - 1831, 2021.
doi:10.14778/3467861.3467871

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/TU-Berlin-DIMA/Condor.

1 INTRODUCTION

An increasing number of applications require low-latency data stream analytics [6, 55]. While this has led to the popularity of big data platforms, such as Flink [12] and Spark [59], reducing the latency remains a relevant problem in the pursuit of real-time data stream analytics [34]. Approximate data analytics based on synopses trade lower latency for a loss in accuracy [25, 32, 39, 49]. These synopses summarize main characteristics from input data while substantially reducing the memory footprint [18], which accelerates data stream analytics with quality guarantees. Many applications benefit from fast approximate answers, e.g, monitoring heavy hitters using count-min sketches [20], maintaining stratified samples using reservoir sampling [57], compressing signals based on Haar wavelets [29], or monitoring network traffic using hyperloglog sketches [22].

However, none of the existing dataflow systems support synopses as pipeline operators, i.e., as first class citizens. The problem is that most of the algorithms maintain synopses in a centralized fashion [39]. As a result, developers of data streaming applications must adapt and implement these algorithms for distributed settings. Besides a good knowledge of both the synopsis algorithm and the data streaming application, the developer needs deep system programming expertise due to its complex architecture. Prior work investigated the integration of synopses into parallel data processing systems [3, 26, 28, 31, 42, 46, 58], but mainly focused on batch or mini-batch processing. It is thus unsuitable for applications with low-latency requirements and does not cover windowed synopses, which are essential for many applications [6, 37].

Integrating synopses into dataflow systems is challenging for many reasons. First, a large variety of data streaming applications exists, all using different kinds of synopses. Creating an abstraction for building synopses has to be sufficiently general while hiding internal processing details and still being easy to use, which is a critical feature of big data systems [5, 12, 59]. Second, the synopsis abstraction must allow for efficient distributed computation. While throughput should scale linearly with the number of nodes, it is hard to achieve due to communication overhead. Third, implementing distributed operations requires significant expertise from users, and hence the synopsis abstraction should hide all these details about distributed operations. Fourth, synopses differ in their algebraic properties. While some of these properties allow for optimizations, others may impose restrictions.

We present Condor, a framework that facilitates the definition of synopsis-based streaming jobs and integrates into any dataflow system that supports window processing. In summary, we make the following contributions.

(1) We define a model that generalizes synopses as stateful window aggregate functions, which allows us to support any one-pass synopsis in a dataflow system (Section 4). The proposed model also classifies synopses based on their algebraic properties.
(2) We devise processing strategies for highly parallel summary construction based on the synopsis' algebraic attributes (Section 5).
(3) We propose a set of evaluation operators that enable efficient processing of windowed synopses from a query stream (Section 6).
(4) We experimentally evaluate Condor's efficiency in terms of throughput and scalability in a distributed system using several queries over four synthetic datasets and one real dataset (Section 7).

Proceedings of the VLDB Endowment, Vol. 14, No. 10 ISSN 2150-8097.
doi:10.14778/3467861.3467871

2 BACKGROUND

We first explain the basic concepts of *synopses* and *windowing*.

Synopses are summarization structures that preserve key properties of the original dataset [18]. Synopses allow for the computation of quantities that are expensive or even impossible to compute precisely and, thus, are a crucial building block for approximate data analytics [25, 39].

The four major families of synopses are samples, histograms, wavelets, and sketches. Sampling techniques are the longest studied synopses with numerous successful applications [3, 9, 26]. Histograms divide datasets into multiple buckets based on values in numerical columns [44]. These are widely studied and incorporated into many commercial relational databases [16, 45]. Conceptually close, wavelets hierarchically decompose numerical columns, maintaining ordering properties of the elements [29]. Lastly, sketches linearly transform the input dataset into a matrix that retains only specific characteristics from the original dataset [18]. Despite their short history, sketches have been already adapted into many different data management applications [1, 11, 19, 20, 43]. We focus on *streaming synopses*, which can be efficiently built with a single pass over the data while not restricting ourselves to a particular synopses family.

Windowing is a processing strategy to organize data stream elements into finite-sized buckets, making windows an effective way to compute analytics over an infinite data stream. There are three main types of windows: tumbling, sliding, and session windows [6, 51]. A tumbling window splits a data stream into non-overlapping segments of the same size, while a sliding window, depending on the slide step, might split a data stream into overlapping segments of the same size. In contrast, a session window covers a period of activity followed by a period of inactivity.

The traditional and most common strategy for window processing is bucketing [38]. This strategy aims at assigning every incoming element to its corresponding windows and outputting the result once the window closes [6, 37]. Most distributed dataflow processing systems, such as Flink, Spark, and Storm, adopt this bucketing technique. More recently, slicing techniques were proposed [13, 35, 36] to optimize window aggregation. They split the input stream into non-overlapping slices of data, triggering a single partial computation per processed element. General Stream Slicing generalizes these techniques and adds out-of-order processing, different window types, and further aggregate functions [52].

3 CONDOR: APPROXIMATE DATA STREAMING ANALYTICS

Condor allows for the specification of synopsis-based streaming jobs on top of general dataflow systems. Condor provides a collection of twelve synopses and an integration to Apache Flink 1.9 [12]. A user either utilizes Condor as a stand-alone java library or defines synopsis-based streaming jobs executed on Flink. Yet, our techniques integrate conceptually into any dataflow engine that supports window processing. Figure 1 illustrates the general architecture of Condor, consisting of two main components: the API and the core. The API provides an abstraction, which the user employs to write synopsis-based streaming jobs. Once done, the user submits

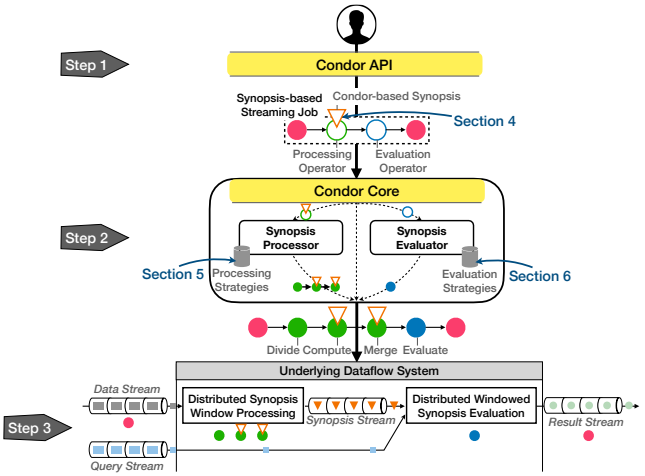


Figure 1: Condor System's general architecture.

these to Condor, where, based on their configurations, the core creates a dataflow pipeline of efficient operators with two processing blocks (the *synopsis processor* and *synopsis evaluator*). Condor then submits this dataflow pipeline to the underlying dataflow system. During the execution, the first processing block (*windowing processing*) is in charge of consuming the input stream and generating the windowed synopses. These are then evaluated by the second processing block (*windowing evaluation*) with a query stream, outputting the approximate results.

We introduce the following example to illustrate how Condor processes a synopsis-based streaming job.

Running Example. Suppose an IP address monitoring application (IP-Job) constantly queries the frequency of varying IP addresses in the input data stream. Typically, the IP addresses are ingested at a high rate. Thus, the IP-Job needs to evaluate the IP address frequencies continuously in sliding windows with a length of 20 seconds and a slide factor of 5 seconds, using count-min sketches [20]. □

To meet these requirements, a developer first writes a synopsis-based streaming job using the Condor API (Step 1, Figure 1). We define a synopsis-based streaming job as a job with an approximate result consisting of two parts: processing and evaluating. To configure these, the user has to provide seven parameters, five of which correspond to the synopses processing:
(a) *data stream*, which is the input (IP addresses in our example).
(b) *synopsis*, which is the desired synopsis class with their initialization values (count-min sketch in our example).
(c) *synopsis output type*, which is either a synopsis per window, *global synopses*, or per window partition pair, *stratified synopses* (global synopses for IP-Job).
(d) *partitioning function*, which indicates how to partition the data stream when outputting stratified synopses (none in our example).
(e) *window type*, which also indicates the windowing parameters (Sliding window in IP-Job).
The other two parameters are for the synopses evaluation:
(f) *query stream*, where each element is an evaluation request (requested IP addresses in IP-Job).
(g) *evaluation function*, which indicates how to process a request (based on the latest window's sketch in our example).

These configurations enable Condor to build an efficient dataflow pipeline that generalizes synopses as stateful window aggregate functions (Section 4). It is worth noting that Condor's API provides an interface to this generalization as well as to our processing strategies, hence, users can scale out new synopses by adding user-defined functions to Condor's execution environment. Besides, our generalization provide the necessary formalism to be adapted to multiple dataflow engines and novel windowing techniques.

Given a developer's job, Condor extracts the job configurations for processing and evaluation (Step 2, Figure 1). The synopsis processor receives the processing configurations and creates a new pipeline using the underlying dataflow engine operators. This new pipeline consists of three phases: (i) a *divide phase* evenly distributes the input data stream workload over all working threads to fully exploit parallelism; (ii) a *compute phase* maintains a partial synopsis in every worker thread, and; (iii) a *merge phase* combines all partial synopses into a single output per window. Each of these phases has different computing strategies, and Condor selects the best of them based on the job's configuration (Section 5).

In turn, the synopsis evaluator receives the synopsis evaluation configuration and selects the best strategies to run the job's evaluation over such per-window synopses. Condor comes with a set of operators, which efficiently evaluate the created synopses with a query stream (Section 6). This feature makes Condor ideal for applications where a synopsis is required for more than one query: Condor creates and maintains a synopsis only once but it can query every synopsis in parallel multiple times. Finally, Condor executes the resulting pipeline on the underlying dataflow system, using native operators and producing approximate results based on the input and query stream (Step 3, Figure 1).

4 SYNOPSIS ABSTRACTION

We model synopses as stateful window aggregate functions. In contrast to current approaches [39], doing so enables us to maintain any one-pass synopsis in a distributed setting as we can use the divide and conquer strategy. We expand our model to classify synopses by their algebraic properties, enabling internal optimizations, such as parallel computation and accepting out-of-order elements. Our model and classification are at the core of Condor's techniques. Without these principles, users have to keep implementing ad-hoc solutions. We discuss our model and classification in the following.

4.1 Synopses as Window Aggregate Functions

Synopses and aggregate functions are conceptually very close to each other. They both aim at reducing a group of values into a single summary value. Thus, it is natural to treat synopses as aggregate functions [30]. However, even as distributed aggregate computation is standard in many systems, e.g., MapReduce [15], the mathematical model in [30] is not enough to formalize how to compute synopses in a distributed setting. This is because not all synopses can be unified in a reduce step. Such aggregate functions are usually maintained in a centralized way, limiting the system's scalability. We tackle this challenge by proposing a distributed window aggregating model to integrate synopses into dataflow systems that support window processing. This computing model consists of three phases: divide, compute and merge.

We define a window W_t as a set of incoming elements $e_i \in I$, where I is the input domain, and $t,\ i \in \mathbb{N}$ are sequential indexes that give the order in which windows are created, and elements arrive. Additionally, we define s^E as a synopsis that has processed all elements belonging to a multiset E, being $E = \emptyset$ the initial state. Suppose we define a synopsis-based streaming job in a system with P_{max} computing cores, where the goal is to maintain global synopses, s^{W_t}, so that each one is built based on all the elements of their corresponding window W_t. To achieve this, the *divide phase* distributes the input data stream among all P_{max} cores, splitting each window W_t into multiple partitioned windows $W_{t,p}$, where p is a partition index. Once every partitioned window is balanced, the *compute phase* incrementally calculates partial aggregates with an *update* function that constructs a synopsis of type S:

$$update_S \colon S \times I \to S, \left(s^E, e\right) \mapsto s^{E \cup e} \tag{1}$$

This *update* function is a stateful aggregate function, which maintains a partial synopsis that is gradually updated as every new element arrives. The compute phase concludes when the synopsis has been updated with all elements of the corresponding window $\{e_i \mid \forall\ e_i \in W_{t,p}\}$ and reaches the state $s^{W_{t,p}}$. Consequently, the system outputs P_{max} partial synopses for every window W_t. Therefore, the *merge phase* has to combine all these partial synopses of type S into a single result for every window, using a *merge* function:

$$merge_S \colon S \times S \to S, \left(s^{E_1}, s^{E_2}\right) \mapsto s^{E_1 \cup E_2} \tag{2}$$

The merge phase concludes once all partial synopses are merged, i.e., $s^{W_{t,0}}.merge(s^{W_{t,1}}) \cdots .merge(s^{W_{t,P_{max}-1}}) = s^{W_t}$. Thus, the system outputs a single synopsis s^{W_t} for every window W_t as required by the synopsis-based streaming job.

Recall that Condor's main goal is to provide a framework that allows for the specification of synopsis-based streaming jobs in general. Hence, the exact implementation of the *update* and *merge* functions for each synopsis is user-dependent and thus is out of the scope of this paper. Yet, Condor provides a collection of twelve synopsis algorithms based on these *update* and *merge* operations (see the first column of Table 1). Users can use our synopses' implementations as examples to integrate new synopses in Condor by implementing these operations as user-defined functions. In this way, it is possible to specialize the core of the code for every specific application. To illustrate this, consider the Running Example in Section 3, where we maintain a count-min sketch [20] for every window. The user selects Condor's count-min sketch, whose implementation keeps the counting matrix as the state of the aggregate function, where every row of the matrix corresponds to a hash function. The *update* function takes every element and increases every cell's count where the hash functions map to, and the *merge* function sums up all the fields from every counting matrices.

Note that the model changes slightly when the user sets up the streaming job to maintain stratified synopses. A stratified synopsis streaming job aims to output a stream of synopses $s^{W_{t,p}}$ for every window partition pair instead of a global synopsis per window. Here, the user can define how to partition the input data stream so that Condor constructs the synopses for every partition separately during the compute phase. The method of partitioning the input data stream can be very beneficial for the estimate's quality. For

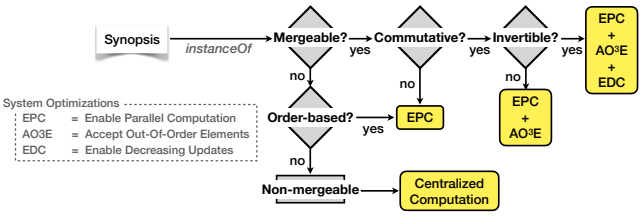


Figure 2: Synopses classification and system optimizations.

example, consider stratified samples [56], which divide the input domain to reduce the sample's variance making the estimate more precise. This technique can also improve the estimates of some sketches, such as the count-min or bloom filter, because in smaller input domain hash coalitions are less probable. Additionally, this also helps answer group-by queries, which is not easy to compute with global synopses and in some cases is not even possible, e.g., counting distinct elements within grouped values using the hyperloglog sketch. Notice that for maintaining stratified synopses, Condor does not have to merge the compute phase's output because their output is already a stream of stratified synopses $s^{W_{t,p}}$.

4.2 Synopses Classification

Synopses have different algebraic properties that Condor exploits to optimize streaming applications. Our synopsis formalization (Section 4.1) enables the system to be aware of these properties by defining five classes of synopses: *mergeable*, *commutative*, *invertible*, *order-based*, and *non-mergeable* synopses. Note that if a user integrates a new synopsis into the system, she has to manually identify to which class it belongs so that Condor is aware of its algebraic properties. The decision tree in Figure 2 illustrates how Condor exploits the synopsis class to decide which synopses may accept out-of-order elements, decreasing updates, or enable parallel computation. We describe these classes below.

Mergeable Synopses: A synopsis is considered mergeable if a *merge* function exists to combine s^{E_1} and s^{E_2} preserving the error and size guarantees [2]. Formally, $s^{E_1}.merge(s^{E_2}) = s^{E_1 \cup E_2}$. Additionally, it is required that both synopses s^{E_1} and s^{E_2} are from the same type and created with the same initialization parameters so that they already have the same error and size guarantees before the merge. Based on this property, Condor enables parallel computation as described in Section 4.1. Most sketches, histograms, and sampling algorithms are mergeable.

Commutative Synopses: Synopses of this class have additionally an *update* function that follows the commutative property, meaning $s.update(e_x).update(e_y) = s.update(e_y).update(e_x)\ \forall x, y$. If this property is available, Condor can accept out-of-order elements during the compute phase as the ordering does not have an influence on these synopses. Otherwise, Condor fallback to using watermarks [6] to allow some disordered elements.

Invertible Synopses: Invertible synopses support the rule of invertibility. Thus, they not only possess an *update* function to increment the partial aggregate, but also a *decrement* function. This function reverts changes to a previous state, $s^E.decrement(e) = s^{E \setminus \{e\}}$. Besides, they also have an *invert* function, which conceptually is the inverse of the *merge* function. Instead of combining two synopses, this method reduces the state from one of the synopses with the actual state of another one, i.e., $s^{E_1}.invert(s^{E_2}) = s^{E_1 \setminus E_2}$. The arrival of out-of-order elements can be problematic for session and count-based windows as it may change multiple window boundaries at processing time [51]. Such cases usually force the system to recompute the window aggregate to generate the correct result, resulting in a performance loss. However, Condor exploits the invertibility property if available, solving inconsistent results with decreasing updates instead of recomputing the whole window.

Order-Based Synopses: A synopsis is order-based if it preserves ordering properties for each of the processed elements. Consider Haar wavelets [29], which are a "lower-resolution" representation of a sequence of processed elements, consuming less space than storing the whole sequence. One can later use this representation to reconstruct the original sequence, as it maintains the ordering and value properties for each processed element. This dependency of order-based synopses on the order of processed items makes them non-mergeable and, therefore, they do not possess a *merge* function that preserves error and size guarantees. Despite this observation, we present a strategy to compute them in parallel and still get a single summary object per window. This strategy consists of computing the partial synopses in parallel as usual with the *update* function, but during the merge phase Condor registers them into a *manager*. We define a manager m^C as an object containing a collection C of partial synopses that we can later use to evaluate indexed queries. To illustrate this, assume we want to answer a sum range query using a manager with a haar wavelet's collection. Each wavelet preserves the indexing information of its processed elements. So once the manager receives a query, it uses the range's indexes to find the wavelets in the collection, containing the information of the elements with such indexes. Then, it forwards a local query to each wavelet and finally sums up the partial results. In this way, such a manager is equivalent to multiple merged synopses, e.g., a global synopsis, as we can use it to evaluate any query and have the same error guarantees as to the contained synopses. However, as the manager contains multiple synopses, the size guarantees are not preserved. We describe the details of evaluating queries with a manager in Section 6.2. Here, we focus on how to adapt a *merge* function for this case, where the output is not a single synopsis anymore but a manager m containing a partial synopsis collection:

$$merge_{M,S}\colon M \times S \rightarrow M, \left(m^C, s^E\right) \mapsto m^{C \cup s^E} \tag{3}$$

By calling this function repeatably, Condor registers all partial synopses $s^{W_{t,p}}$ into a manager that at the end of the computation reaches the state $m^{\{s^{W_{t,p}} \mid \forall\, p \in [0, P_{max}-1]\}}$. Condor now has a single summary object per window, which is equivalent to the output of the merge phase using Equation 3. Notice that, as a result, Condor is the first system to provide an efficient distributed computation and evaluation of order-based synopses.

Non-Mergeable Synopses: To provide a complete solution, we also take synopses into account that do not satisfy any of these properties and, therefore, only have an *update* function. Only in these cases, Condor does not enable parallel computation and computes the synopses centralized in a single compute phase, using the *update* function, i.e., $s^E.update(e_i)\ \forall e_i \in W_t$.

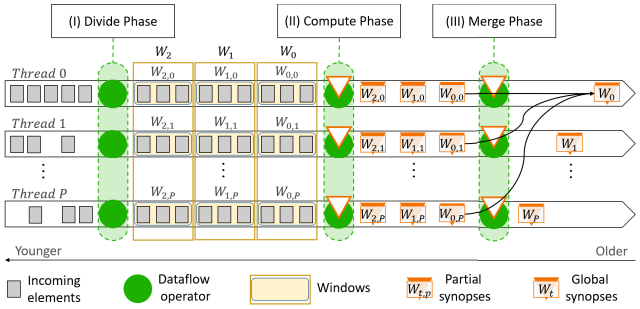


Figure 3: Computing Windowed Summaries Distributed.

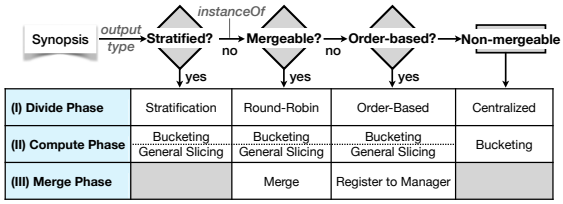


Figure 4: Decision Tree - Processing Strategies.

Condor already provides a collection of twelve synopsis algorithms. We implemented these algorithms based on our distributed window aggregating model and the above synopsis classes. Table 1 maps these synopses to our classification, where *Merg.* stands for mergeable, *Comm.* for commutative, *Inv.* for invertible, and *Ord.* for order-based. Note that a synopsis can belong to several classes, as shown in Figure 2. In such cases, the system first determines if the synopsis is mergeable to enable parallel computation. Then, it checks if the synopsis is commutative to accept or reject out-of-order elements. Additionally, if a synopsis is also invertible, Condor can leverage this property to avoid the complete recomputations if window boundaries change during processing time.

Table 1: Condor's provided synopses - classified.

Synopsis	Merg.	Comm.	Inv.	Ord.
Res. sampling [57]	X			
Fifo sampling	X	X		
Biased res. sampling [4]	X			
Equi-width histogram	X	X	X	
Equi-depth histogram (DDSketch-based) [27]	X	X	X	
Count-min sketch [20]	X	X	X	
Fast AGMS [17]	X	X	X	
Bloom filter [10]	X	X		
Cuckoo filter [21]	X	X		
HyperLogLog [22]	X	X		
DDSketch [40]	X	X	X	
Haar wavelets [32]				X

5 PROCESSING STRATEGIES

Our main goal is to scale out developers' synopsis-based streaming jobs, enabling them to perform their approximate data analytics efficiently. The challenge resides in that scaling them out might not be worth it due to communication overheads. We overcome this problem by reducing communication among distributed nodes [39] and maximizing the usage of the system's resources.

The synopsis processor is the Condor's component that generates the pipeline to maintain the synopses. Figure 3 illustrates this pipeline for distributed synopsis maintenance over windowed data streams. Each horizontal line represents the dataflow pipeline of a thread in the system. Condor splits this pipeline into three phases (*divide*, *compute*, and *merge*) and utilizes different processing strategies in each of them. These strategies are simple by design, as we consider such a simplicity crucial for the applicability of our system. Figure 4 presents an overview of all the processing strategies Condor utilizes depending on the input job's configurations. We discuss these strategies in the following.

5.1 Dividing the Streaming Workload

We distribute the input data stream among all cluster nodes to fully exploit the available parallelism. We assign every element e_i from the input data stream that belongs to window W_t to a partition $p \in [0, P]$. We then split every window W_t into P new sets of elements $W_{t,p}$ that represents the parallel windows belonging to each partition p. In Figure 3, we see that, after the work distribution, each of the three keyed windows (W_0, W_1, W_2) has a bucket containing a subset of elements for every pipeline corresponding to a thread, i.e. $W_{t,0} \cup W_{t,1} \cdots \cup W_{t,P-1} = W_t$. Condor achieves this workload distribution by considering the synopsis class that the user configured to maintain in her input job definition. As shown in Figure 4, the system can utilize four different partition strategies: *stratification*, *round-robin*, *order-based*, and *centralized*.

Stratification. This strategy separates the input data stream for the independent computation of a synopsis per window partition pair. A user must indicate how to divide the input data stream by providing a *stream partitioning function*. Although this strategy gives complete control to users, it might also lead to a performance loss because of idle threads resulting from data skew.

Round-Robin. The system applies this strategy when maintaining mergeable synopses. In such cases, a Round-Robin policy [47] can evenly balance the workload among all P_{max} threads in the system. Typically, dataflow systems offer this workload distribution strategy as an operator [24, 48]. Given a large number of elements per window $|W_t|$, it holds that $|W_{t,p}| \approx |W_t|/P_{max}\ \forall\ p \in [0, P_{max} - 1]$. Thus, the workload is close to optimally balanced among all threads and the system can fully utilize parallel resources.

Order-based. To achieve distributed computation for order-based synopses, Condor first partitions the elements in a way that the order is guaranteed: It uses the Round-Robin strategy [47] but without parallelism. Thus, Condor sends all elements to the same thread and adds them to a buffer. Here, the ordering among the elements is corrected before sending them to the next operator, which has the maximum parallelism again. Once the buffer reaches a predefined threshold capacity and the elements are ordered, Condor dispatches them to the next operator using the Round-Robin strategy. Note that this can become a performance bottleneck if the job has multiple input sources. However, this step is still necessary to ensure a proper ordering of the input elements. If we can not ensure the ordering, then the correctness of the created synopsis can be compromised.

Centralized. If a synopsis is neither mergeable nor order-based, Condor falls back to a centralized computation, which is how current systems maintain synopses [39]. It redirects all the input elements to a single thread that computes the windowed synopses using the corresponding *update* function of that synopsis.

5.2 Computing the Windowed Synopses

After distributing the input data stream, Condor computes the window synopses on each partition. Our formalization does not only work for a single windowing strategy but can also be adapted to new approaches. Condor shows this property by supporting two window processing strategies for computing synopses as windowed aggregate functions: *bucketing* [38], and *general stream slicing* [52]. Condor provides a hybrid solution that adapts the windowing strategy in runtime depending on the window's configuration. It uses the general stream slicing strategy to compute the synopses if the input job requires maintaining sliding- or session-windows. Otherwise, it uses the bucketing strategy.

Bucketing maintains a synopsis for every open window in memory. The system then assigns each input element to their corresponding windows, triggering the synopses' *update* function in each of these windows. This strategy maps perfectly to our synopsis abstraction defined in Section 4. In the case of non-mergeable synopses that are not order-based, the system maintains the synopses in a centralized way using this window processing strategy. Although this strategy is simple and powerful, it triggers redundant computations for overlapping windows, which might significantly hurt the performance.

General Stream Slicing splits the data stream into non-overlapping slices of data in contrast to bucketing. It then assigns each input element to a unique slice, which triggers a single partial computation. Once the window expires, it merges all partial results from all created slices into a final aggregate value and emits it. As a result, the system avoids redundant computations.

We extend the mathematical model from Section 4.1 to support this general stream slicing strategy. We first adapt the general stream slicing strategy to our notation. This strategy splits every window W_t into multiple W_t^l non-overlapping slices of data based on the window type, where $l \in \mathbb{N}$ so that $W_t^l \subseteq W_t$. As Condor partitions the data stream, we end up with a collection of $W_{t,p}^l$, meaning that every element belongs to a unique slice l from a window with an index t, and partition p. Once this is done, Condor computes the synopses $s^{W_{t,p}^l}$ for every partitioned window slice $W_{t,p}^l$ as usual, using their corresponding *update* function (Equation 1). When a window closes, Condor will use the *merge* function (Equation 2) to merge the slice results into partial synopses $s^{W_{t,p}}$, i.e., $s^{W_{t,p}^0}.merge(s^{W_{t,p}^1}) \cdots .merge(s^{W_{t,p}^{L_t-1}}) = s^{W_{t,p}}$ with L_t being the number of slices in which window W_t is divided.

Let $W_{t,p}$ be a window containing a set of streaming elements e_i and each $W_{t,p}^l$ be a subset of the window $W_{t,p}$ so that $\bigcup_l W_{t,p}^l = W_{t,p}$. Then, if the synopses are mergeable, we obtain the same result by updating a synopsis with all the elements of the window $W_{t,p}$ or by updating it separately and merging the slice results. However, this is not the case for order-based synopses, as they are non-mergeable. We overcome this problem by using the *merge* function from Equation 3 to register the slice synopses into a manager. The manager's logic to evaluate queries depends on the order in which Condor processed the elements. As the partition strategy influences this order, we need to be more specific and differentiate the managers' types, depending on how the input was partitioned.

Following this observation, we define two new types of managers. First, the general managers m, whose indexes follow the Round-Robin partition from the workload distribution and; second, the slice managers sm, whose indexes are continuous as the elements in the window were partitioned in continuous slices. Hence, in the case of computing order-based synopses using the general stream slicing strategy, Condor registers all slice synopses $s^{W_{t,p}^l}$ into a slice manager using the *merge* function from Equation 3. The output contains then of a slice manager $sm^{C_{t,p}}$ for every window partition pair, with $C_{t,p} = \{s^{W_{t,p}^l} \mid \forall\, l \in [0, L_t - 1]\}$. In this way, the merge phase can receive a single object per partition as the bucketing strategy.

The general stream slicing strategy also supports out-of-order processing by exploiting invertibility or commutativity if available. Our classification from Section 4.2 indicates to the system, which algebraic properties are available for the current synopsis type.

5.3 Merging the Partial Summaries

Once Condor computes the partial synopses for each partition, it performs a union of these synopses if it is possible and global synopses are required. Condor performs the merge phase without parallelism by either *merging* the partial synopses into a global synopsis or *registering* these at a manager.

Merge. If the system is maintaining mergeable synopses, Condor simply merges the partial synopses $s^{W_{t,p}}$ the compute phase generates, using a *merge* function (Equation 2). The final output is a single synopsis s^{W_t} per window as illustrated in Figure 3.

Register at Manager. A synopsis manager's role is to store all partial synopses as an indexed collection to redirect queries into their corresponding partition. This mechanism makes it is possible to compute order-based synopses separately, which enables parallel processing. Thus, in the case of using the bucketing strategy in the compute phase, Condor uses the *merge* function (Equation 3) to register all partial synopses into a general manager. Hence, the output are multiple managers $m^{\{s^{W_{t,p}} \mid p \in [0, P_{max}-1]\}}$ one per window. However, if general stream slicing is used, Condor cannot use the *merge* function as the compute phase's output consists of slice managers instead of synopses. Still, as a manager evaluates any query preserving error guarantees, Condor can register them into a general manager. When a query arrives, the general manager forwards the query to the corresponding slice manager, which, in turn, forwards the query to the corresponding partial synopsis for their evaluation. Therefore, we define a new function to register slice managers of type SM into a general manager of type M:

$$merge_{M,SM} \colon M \times SM \to M, \left(m^C, sm^{C'}\right) \mapsto m^{C \cup sm^{C'}} \quad (4)$$

Thus, the output in this case are then general managers $m^{\{sm^{C_{t,p}} \mid p \in [0, P_{max}-1]\}}$. Hence, the merge phase's output using

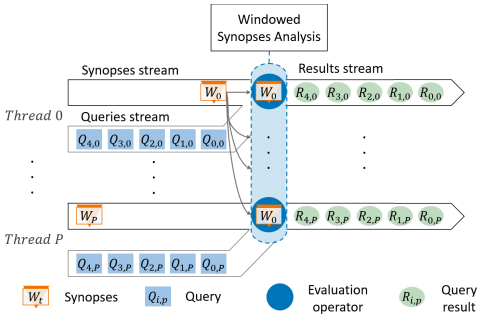


Figure 5: Evaluating Windowed Synopses.

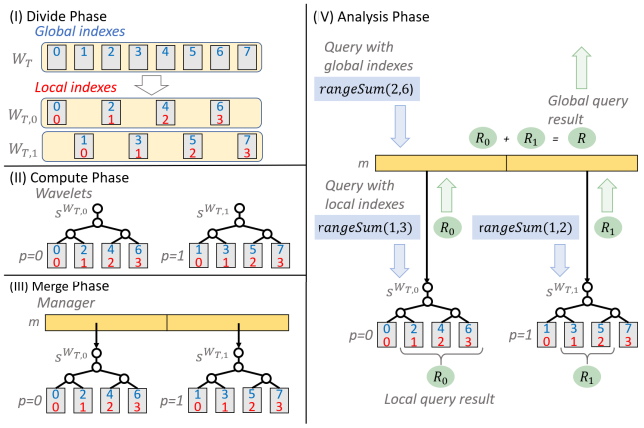


Figure 6: Order-Based Synopses Evaluation – Wavelets.

general stream slicing is equivalent to the corresponding output when using bucketing: both preserve the same error guarantees.

6 EVALUATION STRATEGIES

When the user sets the job to maintain global synopses, Condor performs the merge phase without parallelism. We thus need to find a way to make these global synopses available to all threads to evaluate them efficiently. Furthermore, querying synopses is a challenge because we need means to represent queries to allow the system to evaluate the synopses accordingly. We overcome these challenges by introducing two *evaluation operators*. However, users can alternatively create their operators based on the API of the underlying dataflow engine. For example, they may use a map operator that extracts the top-k elements from each windowed count-min sketch constructed by the streaming job.

6.1 Evaluating Windowed Synopses

We leverage that the most recent streaming data suffices to evaluate a query over sliding windows [9]. Thus, we define two different two-input stream operators with broadcast state [23], namely *QueryLatest* and *QueryTimestamped*. Figure 5 illustrates how these operators work. The broadcast state strategy is used for streaming applications where a low-throughput event stream, in our case, the synopsis stream, is broadcasted to all parallel instances of an operator. The system then evaluates the event stream against all the elements coming from the query stream. This way, the system can query the same synopsis simultaneously on every thread, even as each synopsis was constructed only once.

QueryLatest. Let BS_{ql} be the broadcast state of the *QueryLastest* operator. When a window with index T is closed, the system broadcasts synopsis s^{W_T} to all computing cores in the system, which run an instance of the *QueryLastest* operator, i.e., $BS_{ql} = \{s^{W_T}\}$. The system can then evaluate the queries on the same synopsis with maximum parallelism. Consider our running example (Section 3), where each query of the query stream contains a requested IP address. All *QueryLatest* operator instances simultaneously evaluate these queries with the count-min sketch of the latest window, outputting the requested IP addresses' approximate frequencies.

QueryTimestamped. In contrast to *QueryLatest*, the system now broadcasts a collection of synopses of the last user-defined N synopses, i.e., $BS_{qt} = \{s^{W_t} \mid \forall\, t \in [T-N, T]\}$, where BS_{qt} is the broadcast state of the *QueryTimestamped* operator. When a *QueryTimestamped* instance receives a new synopsis, it deletes the oldest synopsis $s^{W_{T-N}}$ and adds the new one. This operator also requires each query to have a timestamp, q_{ts}, which is used to search for all the synopses in BS_{qt} that contain that period of time in their window definition. Therefore, as windows may overlap, a single query may generate multiple results.

In the case of handling stratified synopses, each operator maintains P times more synopses in their broadcast state, where P is the total number of partitions, i.e., $BS_{ql} = \{s^{W_T,p} \mid \forall\, p \in [0, P-1]\}$ for *QueryLatest*, and $BS_{qt} = \{s^{W_t,p} \mid \forall\, t \in [T-N, T], p \in [0, P-1]\}$ for *QueryTimestamped*. Therefore, each query also requires a partition value p, so that Condor can evaluate it only with the synopses corresponding to the same partition p.

6.2 Evaluating Order-Based Synopses

Here, the evaluation operators receive a stream of managers as input instead of a stream of synopses. Therefore, the user must indicate how the manager forwards the evaluation requests to the corresponding synopses, depending on the partition strategy. Condor provides users with a basic manager implementation so that they do not have to provide the logic of how partial synopses are stored. This way users can implement their own manager by extending this basic manager. Condor provides all functions to get the corresponding synopsis based on the indexed query.

Figure 6 gives an example illustrating how Condor evaluates order-based synopses using a manager and query forwarding. Consider we want to construct Haar wavelets using parallelism of two. First, Condor divides and balances the incoming stream using the Round-Robin partition [47] (Phase I). After the workload distribution, the compute phase (Phase II) redirects all elements with $i \bmod P_{max} = p$ to thread p to construct the partial synopses. Then, the merge phase (Phase III) unifies all the partial synopses by registering them into a *manager* m. During the evaluation phase (Phase IV), we then calculate the range-sum for all values between indexes $left$ and $right$ in window W_T, with $left = 2$ and $right = 6$. In this case, the manager m asks every partial result $s^{W_T,p}$ to calculate a range-sum using local indexes instead. As we

assumed a Round-Robin partition, m calculates the local indexes in a partition as $\lfloor index/P_{max} \rfloor$. However, it is necessary to check if the local index plus the partition value p maps to an index inside the original range between $[left, right]$. If the mapping does not fall inside the given range, manager m corrects the local index by adding one for a left index or subtracting one for a right index. The result for this example is $m^{\{s^{W_{T,0}}, s^{W_{T,1}}\}}.rangeSum(2,6) = s^{W_{T,0}}.rangeSum(1,3) + s^{W_{T,1}}.rangeSum(1,2)$. This technique also work for any range or point query. This is possible as order-based synopses preserve the element's indexing properties: Users only redirect the query to the corresponding synopsis using the indexes.

7 EVALUATION

We evaluate the most important properties of Condor and compare it to one-off implementations and Yahoo's DataSketches [58]. We carried out these experiments with four main questions in mind: (i) How efficient is Condor compared to the baselines (Section 7.2)? (ii) Do our parallel processing strategies maintain the accuracy guarantees compared to centralized approaches (Section 7.3)? (iii) Do our processing strategies scale linearly to the total number of cores in the system (Section 7.4)? (iv) How does distributed processing compare to single-threaded processing (COST [41]) (Section 7.5)?

7.1 Experimental Setup

Setup. We implemented Condor on top of Apache Flink 1.9 [12] and used the Scotty [53] libraries for the general stream slicing strategy. Condor is part of the Agora project [54]. We performed our experiments on a cluster of 22 machines: The job-manager had 48 GB of memory and 16 processing cores with 2,40 GHz; The task-manager had 32 GB of memory and 16 processing cores with 2.0 GHz. All machines have a 1Gbps Ethernet connection and Ubuntu 18.04. We configured all JVMs with a heap memory size of 28 GB. We report the average throughput. For our measurements, we implemented the Yahoo Streaming Benchmark for Flink [14].
Baselines. We compare Condor features to two one-off implementations of the count-min sketch. The goal of both implementations is to maintain global windowed synopses. The first one is named "Flink centralized", where we use global windows and an aggregate function to compute the sketches in a single thread. The second one, called "Flink distributed", uses keyed windows to partition the windows and merges the partial sketches afterward. For maintaining sketching algorithms in a distributed environment, we compare Condor with Yahoo's DataSketches [58]. Finally, we compare the runtime of our distributed implementation to a basic single-core implementation without any overhead of an underlying system.
Datasets. We employed five different data sources. The first four are synthetic, which are *UniformDataset*, *NormalDataset*, *ZipfDataset*, and *IPDataset*. The keys: of the first one follow a uniform distribution with values between $[1, 1000]$; of the second one follow a Normal distribution with $\mu = 1000$ and $\sigma = 300$; of the third one follow a Zipf distribution with $a = 1.1$; and of the fourth one are uniformly random 32-bit integers as one can encode an IP address as a 32-bit integer. The fifth data stream is the *NYCTaxiDataset* [50].
Queries. As no precise real-world queries are yet using windowed synopses, we defined synthetic queries to test our system. We consider two classes of queries: *compute-* and *evaluate-jobs* to deal with synopses. For the experiments, we created multiple compute jobs only to create windowed synopses using different configurations to vary the system's maximum parallelism, synopsis' output type, stratification degree, and window type. Besides, we extend these jobs to be evaluate-jobs running real-world queries based on the NYCTaxiDataset. Additionally, we implemented our running example in Section 3 (IP-job) as an evaluate-job to query count-min sketches constructed with the IPDataset. Table 3 gives an overview of the initialization parameters for each synopsis type.

7.2 Efficiency

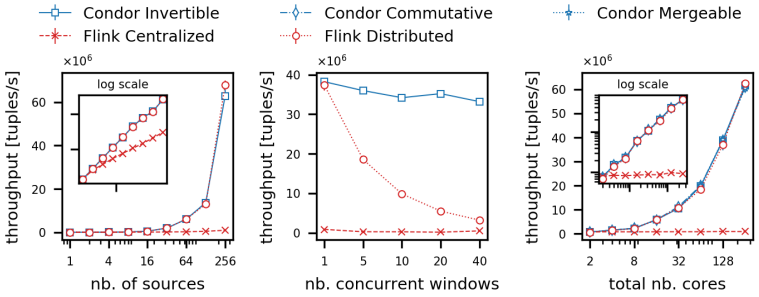
We evaluate the efficiency of Condor from three perspectives. First, we measure the efficiency of Condor's features compared to the one-off custom implementations of the count-min sketch in Flink (Section 7.2.1). Second, we compare Condor's sketch libraries with the ones in Yahoo! DataSketches [58] (Section 7.2.2). Last, we evaluate if Condor can indeed boost the performance of Yahoo! DataSketches by integrating their implementation as a user-defined function based on our API and synopsis abstraction (Section 7.2.3).

7.2.1 System Features. We start by evaluating the different features of our system by comparing it with our one-off count-min sketch Flink implementations. These features are serialization/deserialization of the input data stream, computation with overlapping windows, and the synopses classification.

Results. Figure 7a illustrates the results of the serialization evaluation. Note that for this evaluation, we maintained the system parallelism at 256 and gradually increased the number of sources that simultaneously ingest data into our pipeline. We observe that in cases where we have a single source, the serialization cost is so high for the distributed approaches that there is no significant improvement over the centralized approach. However, as soon as we increase the number of sources, the cost per tuple decreases, which makes the serialization pay off: We observe that the distributed approaches achieve a throughput twice higher than Flink centralized from four sources; Especially, we observe they outperform Flink centralized by 75× for 256 sources.

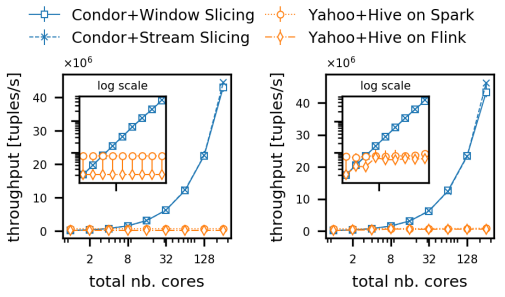
Figure 7b shows the results of our second set of experiments, where we tested the performance for an increasing number of concurrent windows. For these experiments, we set the parallelism to 256 as well as the number of sources. The results show the benefits of combining different windowing strategies. We observe that the only case where a Flink Distributed has a similar performance as Condor is when there are no overlapping windows. However, the performance of Flink Distributed drastically drops as soon as the streaming job has more overlapping windows. On the other side, Condor switches to general stream slicing when windows start to overlap, making it much more stable even with an increasing number of concurrent windows. In contrast to the bucketing strategy, the general stream slicing strategy assigns every element to one single slice of non-overlapping data instead of multiple windows. In more detail, the bucketing strategy computes an update for every element in the synopsis of each concurrent window, while general stream slicing triggers a single update per element.

We also evaluated the importance of our synopsis classification from Section 4.2. For these experiments, we varied the number



(a) Serialization. (b) Concurrent windows. (c) Classification.

Figure 7: Comparison to custom one-off implementations.



(a) A global sketch. (b) A sketch per partition.

Figure 8: Hyperloglog sketches performance.

of cores and considered three different streaming jobs for Condor: in the first one, we used the count-min implementation as an invertible synopsis; in the second one, we made our count-min sketches only commutative; and in the last one, we set count-min sketches as only mergeable synopses. Figure 7c shows the results. We observe that the performance of all three variations and the distributed Flink implementation are very similar. However, the difference comes when comparing the estimation results with an input source where its elements are not always in order. To show this, we ran the experiment using a data stream where ten percent of the elements has a much smaller timestamp than expected. This forces the system to deal with out-of-order elements. We observed that the Flink implementation uses watermarks to deal with this problem. Nevertheless, as the out-of-order elements have a much smaller timestamp than expected, Flink and Condor's mergeable synopsis implementation ignores those elements causing the sketch to be incomplete and hence their estimations inaccurate. In contrast, Condor's invertible and commutative implementations can update the corresponding sketch even if elements arrive late, ensuring better estimates. However, this holds only for time-based windows because, as soon as we switch to session or count-based windows, only the invertible implementation provides a correct result. This is because an element's late arrival can change the window boundaries forcing the system to recompute those windows. Thus, Condor uses decreasing updates to change the windows' state without any need or re-computation.

Summary. Condor outperforms one-off custom implementations: In the worst case, it performs equally good as a one-off distributed implementation; It preserves its high performance even in scenarios with a high number of concurrent windows; Additionally, it leverages our synopsis classification to deal with out-of-order elements improving the estimation results.

7.2.2 Synopses Libraries. We now evaluate the efficiency of our sketch libraries and compare it with the Yahoo! DataSketches libraries [58]. Yahoo! DataSketches provide java libraries to create sketches in every java-based program and additionally offers connectors to Apache Hive [7] and Pig [8]. Note that these systems do not provide real-time stream processing. We decided to compare our hyperloglog sketch (HLL) [22] implementation with Yahoo's HLL implementation to see which implementation makes better use of the system's parallelism. For this, we configured the hyperloglog

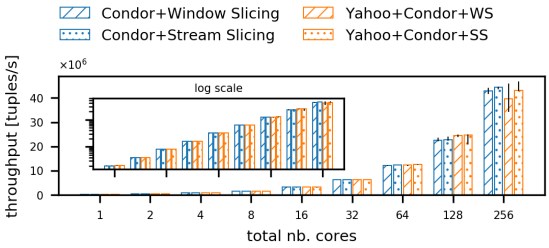


Figure 9: Performance Yahoo! DataSketches with Condor.

compute-job to maintain a single sketch for the whole dataset (i.e., a global sketch), and then we configured the jobs to compute a sketch per partition (i.e., stratified sketches). The number of partitions is the same as the available number of cores in the system.

Results. In Figures 8a and 8b, we observe that our processing strategies significantly outperform Yahoo. Yahoo's implementation outperforms Condor in scenarios with very low parallelism (below eight cores) because of Flink's system overhead. However, we observe that already with two quadcore computers, our system achieves better performance. More interestingly, we observe that Condor's HLL implementation scales linearly with the number of available cores and outperforms Yahoo by more than one order of magnitude (46× faster). It can achieve this due to its divide and conquer design, making better use of the system's parallelism. In particular, we observe that Yahoo's implementation computes the global sketches centralized (Figure 8a). Hence, even in the log-scale image, we cannot observe any significant change in their performance as we increase the parallelism degree. We observe the same behavior for Yahoo's implementation while computing a sketch per partition using Hive on top of Spark (Figure 8b). However, when using Hive on top of Flink, we see some performance improvement because Flink tries to group the partitions on different cores. After reaching a parallelism of 16, the throughput remains constant. This indicates that the system overhead is more significant than any possible improvement of having more available cores.

Summary. Condor's implementation linearly scales with the total number of cores in the system, showing that our implementation is specifically designed for high parallelism applications. This is not the case for Yahoo! DataSketches using Hive.

7.2.3 Integrating User-Defined Synopses. An essential feature of Condor is that it allows users to implement their synopses via a simple API: Users can focus on the application logic instead of intricate internal details. We tested this feature by adapting Yahoo's HLL sketch implementation to our API, showing that Condor enables any mergeable synopsis to scale linearly with the parallelism.

Results. Figure 9 illustrates the results of this experiment. We see that the improvements in Yahoo's HLL sketch's scalability and performance when using Condor are remarkable. Now Yahoo's HLL sketch performance is very similar to Condor's original implementation. More importantly, Yahoo's HLL sketch now scales linearly with the system's parallelism (see log scale plot). Note that achieving this was easy as Condor's API is simple to use. This shows both the ease-of-use of Condor and its power to provide a scalable processing environment for any synopsis.

Summary. We have proven the flexibility of our system by integrating Yahoo's HLL sketch on Condor with our API. With this integration, performance and scalability improve significantly. Thus, we can say that Condor is the perfect complement to Yahoo! DataSketches libraries in real-world applications.

7.3 Reliablility

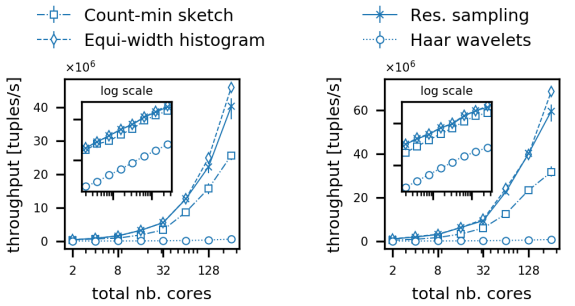
Mergeable synopses preserve the error and size guarantees after being merged with other synopses of the same class [2]. Recall that error guarantees may not be exact as synopses typically provide a theoretical error range [18]. Nevertheless, we now demonstrate that our parallel processing strategies also preserve the error and size guarantees. We performed our accuracy experiments with two different configurations: a 256 parallelism degree and a centralized setup. We then evaluate the approximate results of both configurations with the real results. For these experiments, we used the NYCTaxiDataset as input and defined five evaluate jobs to create a single global synopsis from the input data stream. Each of these jobs is inspired by a real-world query for testing five different synopses: (i) Get the number of entries in the dataset of each taxiID using the count-min sketch; (ii) Predict the number of distinct taxiID's using the hyperloglog sketch; (iii) Get the number of taxi rides with start longitude between $[-73.991119, -73.965118]$ using an equi-width histogram; (iv) Compute the average passenger count using a reservoir sample, and; (v) Compute the range sum for every 10,000 entries using one-pass Haar wavelets.

Results. Table 2 shows the results. We observe that the relative accuracy of the centralized and distributed setups is the same for all synopses, except for the reservoir sampling (iv). The relative accuracy is slightly better for the distributed setting. However, this difference is not significant, as both results are within the expected variance ranges. Concerning the size guarantees, we observe that for all the mergeable synopses (i, ii, iii, iv), the resulting global synopsis size from each distributed setup is the same as the original size. Still, this does not hold for Haar wavelets as they are order-based synopses, for which Condor does not merge the partial synopses but registers them into a manager. Given a parallelism degree of 256, the manager is about 256 times bigger than a single synopsis.

Summary. Our distributed processing strategies preserve error and size guarantees for all mergeable synopses and only the error guarantees for order-based synopses.

Table 2: Condor's relative accuracy and synopsis size ratio.

Synopsis	Centralized	Distributed	Size ratio to centralized
(i) Count-min	0.0015	0.0015	1.0
(ii) Hyperloglog	0.1169	0.1169	1.0
(iii) Equi-width hist.	0.0169	0.0169	1.0
(iv) Res. sampling	0.0071	0.0063	1.0
(v) Haar Wavelets	0.0344	0.0344	~256.0



(a) Bucketing. **(b) General stream slicing.**

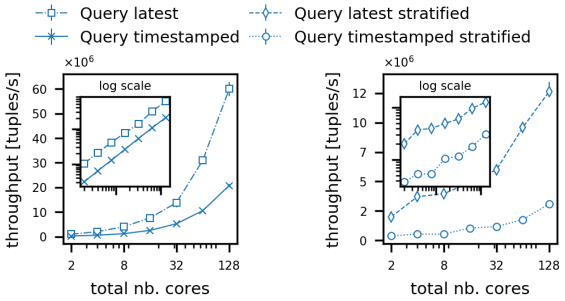
Figure 10: Scalability for synopsis computation.

7.4 Scalability

One of our main motivations for Condor was to scale gracefully with the number of available cores in the system. We now focus on evaluating the scalability of Condor's processing strategies when increasing the degree of parallelism. Additionally, we studied how different data sources can influence the performance of Condor.

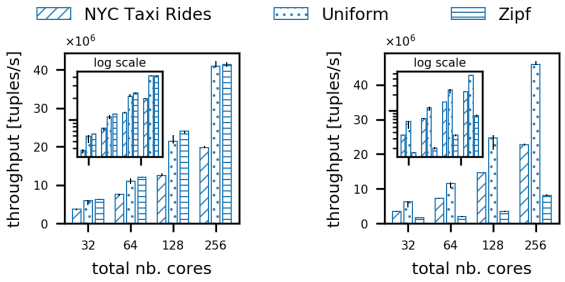
7.4.1 Synopses Processing Scalability. First, we analyze Condor's throughput when computing windowed synopses as we increase the degree of parallelism. We use the compute-jobs for constructing count-min sketches [20], equi-width histograms, reservoir samples [57], and one pass Haar wavelets [32]. On all of these configurations, we want to obtain a global synopsis per window.

Results. Figure 10 illustrates the results: Figure 10a shows the results when using the bucketing strategy, and Figure 10b shows the results when using the general stream slicing strategy. Overall, we observe that all of Condor's synopses scale linearly with the number of available cores in the system, even the one pass Haar wavelet, as seen in the log scale figures. Still, we observe a different performance between the Haar wavelet and all the other synopses. This performance gap occurs because Haar wavelets are order-based synopses, and hence they use a buffered round-robin partitioning without parallelism in the divide phase. In contrast, all the other synopses are mergeable, and hence they use the round-robin partition with maximum parallelism. However, we can also observe a slight performance difference among these mergeable synopses. This is because every synopsis has a different implementation of the update and merge functions (Section 4.1). Therefore, even if the processing pipeline is the same, the code's core can differ. For example, the count-min sketch performs worse than the equi-width histogram because its update function must first process each row's



(a) Global synopses. (b) Stratified synopses.

Figure 11: Scalability for synopsis evaluation.



(a) Global synopses. (b) Stratified synopses.

Figure 12: Performance with different data sources.

hash functions and then increment the corresponding counters; the equi-width histogram, instead, selects the correct bucket and then increases a single counter.

Summary. All Condor's synopses scale linearly with the number of available cores no matter which family they belong to.

7.4.2 Evaluation Operators Scalability. We proceed by analyzing the throughput of Condor's evaluation operators as we increase the degree of parallelism. For these experiments, we used the count-min sketch evaluate-job, which is the exact implementation of the IP-job (Running Example in Section 3). Accordingly, we generated a stream of queries containing random 32-bit integers, representing frequency requests for specific IP addresses. Each job contains a different evaluation operator presented in Section 6). We set up the QueryTimestamped- and QueryTimestampedStratified operator to maintain the last 120 synopses per partition in the broadcasted state. For the stratified jobs, we configure the number of partitions to be the same as the parallelism degree.

Results. Figure 11a shows the results when evaluating global synopses. We observe that both operators scale linearly with the number of cores, however, the QueryLatest operator a better performance than for the QueryTimestamped operator. QueryLatest maintains a single synopsis in the broadcasted state, while QueryTimestamped maintains more than one synopsis (120 with these configurations). Figure 11b shows the results when evaluating stratified synopses. Here, it can be observed, that the operators do not scale as smoothly as for the global synopses because the broadcasted state regularly gets bigger as the number of partitions is increased. This does not occur when evaluating global synopses as Condor maintains the same amount of synopses (one for QueryLatest and 120 for QueryTimestamped) in the broadcasted state, no matter the parallelism. On the other side, we configured the number of partitions to be the same as the parallelism degree for the stratified jobs. This means that for each new partition, we maintain 120 more synopses in the broadcasted state. Thus, if the number of partitions is set to a constant value, our evaluation operators would linearly scale on stratified as on global synopses.

Summary. All the Condor's evaluation operators scale linearly to the number of available cores: the smaller the broadcast state, the better their performance.

7.4.3 Data Sources. We now test the performance of our synopsis processing strategies using different datasets. We especially want to test our strategies under data skew. Therefore, these tests use two syntectic datasets: one without data skew, which is the UniformDataset, and; one with very high data skew, which is the ZipfDataset. However, to confirm our results, we also performed our experiments with a real-world dataset, the NYCTaxiDataset [50]. We used the count-min sketch compute-job with two different outputs in these experiments: global and stratified-synopses.

Results. Figure 12 shows the results. Overall, it can be observed, that when computing global synopses (Figure 12a), the data distribution does not have any effect on the scalability of our system. This is confirmed by the log scale figure, showing that the throughput scales linearly for any of the datasets. This is because of the round-robin partitioning that Condor employs in the divide phase (Section 5.1). This strategy always balances the workload among all cores in the system regardless of the datasets' distribution. However, this is not the case when computing stratified synopses (Figure 12b). We observe that data skew does influence the performance of our system. This is because Condor uses the stratification strategy in the divide phase for such cases. In this strategy, the user decides how to partition an incoming data stream depending on each tuple's content. In these experiments, we configured to split the datasets based on their key attribute. Thus, in contrast to UniformDataset, the ZipfDataset causes uneven partitioning as it has few groups of keys that contain a larger number of elements. As a result, the merge phase has to wait longer for these big groups, causing back-pressure and inefficient computation. Note that solving this data skew problem is out of the scope of this paper. Finally, we also observe that the performance with the NYCTaxiDataset is worse than the others. This is because each tuple of the NYCTaxiDataset is much bigger (75 bytes/tuple) than the tuples of our synthetic datasets (16 bytes/tuple): The quantity of bytes transferred among operators is then almost five times bigger.

Summary. The workload distribution is critical for performance: the more evenly the data partitioning, the better the performance.

7.5 COST

We measured the COST (Configuration that Outperforms a Single Thread) [41] of all the synopses that Condor provides (Section 4.2). Our goal in these experiments is to study the feasibility of using each

Table 3: Condor's synopses COST.

Synopsis	Global	Stratified	Parameters
Res. sampling [57]	8	32	size=10000
FIFO sampling	32	14	size=10000
Biased res. sampling [4]	6	12	size=10000
Equi-width histogram	8	32	nBuckets=1000
Equi-depth histogram [27]	70	40	nBuckets=1000; α = 0.1
Count-min sketch [20]	8	10	w=65536; d=5
Fast AGMS [17]	5	10	w=65536; d=5
Bloom filter [10]	6	64	k=4; m=128
Cuckoo filter [21]	12	32	f=8; b=8; m=2^15
HyperLogLog [22]	7	16	m=4096
DDSketch [40]	6	64	α = 0.05; m=256
Haar wavelets [32]	32	4	N=10000

synopsis in real-world applications. We used two different compute jobs for each synopsis: one where the goal is to compute global synopses per window and one that maintains stratified synopses. We can find the initialization parameters for each synopsis in (the fourth column of) Table 3. For the stratified jobs, we used 256 partitions. We also implemented a single-thread version of each of these jobs without any system overhead to better understand how many cores our processing strategies require to pay off.

Results. Table 3 shows the results of these experiments, where the green cells denote a very low COST ($\leq$ 16), the yellow cells a middle COST ($<$ 64), and the orange cells a high COST ($\geq$ 64). On the table's second column, we find the synopsis' COST in a global configuration. As we can see, the COST of each synopsis depends directly on the synopsis type (see the first column). This is caused by the variety of the update and merge functions. If the synopsis has an efficient merge function, the COST in the global configuration is low. This is the case of nine of the synopses in our collection (green background), where the COST is lower than (or equal to) 12 cores, making them ideal for almost every real-world application. However, we have a higher COST for FIFO sampling and Equi-Depth histograms because their merge functions are inefficient. This injects a significant overhead, which makes it hard for distributed computation to pay off. Another synopsis with a high COST is the Haar wavelet. However, in contrast to FIFO sampling and Equi-Depth histograms, Haar wavelet suffers from order-based synopses, which require an order correction in the divide phase (Section 5.1). It does so without parallelism, causing a bottleneck in the pipeline, which, in turn, hurts the job's performance. However, its update function is so expensive that with a parallelism of 32, the overhead injected in the divide phase is not significant anymore.

Regarding the COST of the stratified configuration (third column), we again observe that the results vary depending on the synopsis used. However, in contrast to the global configuration setting, synopses with expensive merge functions are favored. This is because a stratified job does not have a merge phase. In particular, we observe that FIFO sampling and Equi-Depth histograms reduced their COST by approximately half. Nevertheless, the other synopses, which have very efficient update functions, now have a higher COST than before. This is because the overall system's overhead is too high compared to the time we could gain with the distribution. In other words, the more efficient the update function, the higher the COST will be in a stratified configuration. Finally, we also observe that Haar wavelets have different behavior than the other synopses. This is caused by the expensive update function a Haar wavelet requires, causing poor performance for maintaining multiple synopses in a single core.

Summary. The efficiency of Condor depends on the update and merge functions. However, Condor's synopses reach high throughput with only a few cores: mostly less than 12 cores on a global setup and less than 32 in a stratified setup. We can thus conclude that Condor's synopses are suitable for real-world applications.

8 RELATED WORK

Yahoo! DataSketches [58], BlinkDB [3], StreamApprox [46], SnappyData [42] are good representative of employing summarization algorithms into large scale environments that must handle big data. Each of these systems offers approximate analysis via synopses. Condor has clear differences to them, including important new features that were not previously covered. Only StreamApprox supports real-time processing with synopses as our approach, while BlinkDB, Yahoo! DataSketches, and SnappyData support only batch or mini-batch processing. Besides, BlinkDB and SnappyData use sampling and sketches for approximate analysis. However, both employ synopses only as internal optimizations rather than available processing blocks for approximate data analysis applications. Moreover, StreamApprox offers only sampling as a pipeline operator, giving the user more control over the data processing. Yahoo! DataSketches provide java libraries to create synopses in every java-based program. Condor, in contrast, offers a much broader collection of synopses with representatives of every class of synopses, i.e., sampling, sketches, histograms, and wavelets. Condor is the only framework that provides an API to easily define new synopsis algorithms and the infrastructure for their maintenance. The Scotch [33] framework also features an aggregation-based programming model. However, it is strictly limited to sketch synopses and neither covers distribution nor windowing.

9 CONCLUSION

We presented Condor, a framework for the specification of synopsis-based streaming jobs on top of general dataflow systems. Condor lays down the mathematical foundations for efficient synopses computation in distributed streaming applications. It is the only framework that equips streaming applications with parallel computation and evaluation for all synopsis types, including sketches, histograms, samplers, and wavelets. Its techniques offer high-throughput for parallel synopsis maintenance while preserving accuracy guarantees of centralized techniques. We evaluated its effectiveness using multiple representative jobs with four synthetic and one real dataset. We show that it significantly outperforms existing approaches, up to a factor of 75x, due to better resource utilization and less data transfer among nodes. The performance of all its techniques scales linearly with the parallelism degree.

ACKNOWLEDGMENTS

This work has received funding from the German Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (01IS18025A and 01IS18037A) and Software Campus (01|S17052).

REFERENCES

[1] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2015. Profiling relational data: a survey. *The VLDB Journal* 24, 4 (2015), 557–581.

[2] Pankaj K Agarwal, Graham Cormode, Zengfeng Huang, Jeff M Phillips, Zhewei Wei, and Ke Yi. 2013. Mergeable summaries. *ACM Transactions on Database Systems (TODS)* 38, 4 (2013), 26.

[3] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 29–42.

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

[5] Divy Agrawal, Sanjay Chawla, Bertty Contreras-Rojas, Ahmed K. Elmagarmid, Yasser Idris, Zoi Kaoudi, Sebastian Kruse, Ji Lucas, Essam Mansour, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, Saravanan Thirumuruganathan, and Anis Troudi. 2018. RHEEM: Enabling Cross-Platform Data Processing - May The Big Data Be With You! *PVLDB* 11, 11 (2018), 1414–1427.

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

[7] Apache Software Foundation. 2020. *Apache Hive*. https://hive.apache.org/

[8] Apache Software Foundation. 2020. *Apache Pig*. https://pig.apache.org/

[9] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. 2002. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 1–16.

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

[11] Andrei Broder, Michael Mitzenmacher, and Andrei Broder I Michael Mitzenmacher. 2002. Network applications of bloom filters: A survey. In *Internet mathematics*. Citeseer.

[12] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[13] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. 2016. Cutty: Aggregate sharing for user-defined windows. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 1201–1210.

[14] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. 2016. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 1789–1792.

[15] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce online.. In *Nsdi*, Vol. 10. 20.

[16] Graham Cormode, Antonios Deligiannakis, Minos Garofalakis, and Andrew McGregor. 2009. Probabilistic histograms for probabilistic data. *Proceedings of the VLDB Endowment* 2, 1 (2009), 526–537.

[17] Graham Cormode and Minos Garofalakis. 2005. Sketching streams through the net: Distributed approximate query tracking. In *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 13–24.

[18] Graham Cormode, Minos Garofalakis, Peter J Haas, and Chris Jermaine. 2012. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases* 4, 1–3 (2012), 1–294.

[19] Graham Cormode and Marios Hadjieleftheriou. 2008. Finding frequent items in data streams. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1530–1541.

[20] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[21] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM, 75–88.

[22] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*. Discrete Mathematics and Theoretical Computer Science, 137–156.

[23] Apache Flink. 2020. The Broadcast State Pattern. https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/state/broadcast_state.html

[24] Apache Flink. 2020. *Physical Partitioning*. https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators/#physical-partitioning

[25] Minos N Garofalakis and Phillip B Gibbons. 2001. Approximate Query Processing: Taming the TeraBytes.. In *VLDB*. 343–352.

[26] Phillip B Gibbons, Yossi Matias, and Viswanath Poosala. 1997. *Aqua project white paper*. Technical Report. Technical report, Bell Laboratories, Murray Hill, New Jersey.

[27] Phillip B Gibbons, Yossi Matias, and Viswanath Poosala. 2002. Fast incremental maintenance of approximate histograms. *ACM Transactions on Database Systems (TODS)* 27, 3 (2002), 261–298.

[28] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D Nguyen. 2015. Approxhadoop: Bringing approximations to mapreduce frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. 383–397.

[29] Alfred Haar. 1909. *Zur theorie der orthogonalen funktionensysteme*. Georg-August-Universitat, Gottingen.

[30] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. 2014. A survey of distributed data aggregation algorithms. *IEEE Communications Surveys & Tutorials* 17, 1 (2014), 381–404.

[31] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. 2016. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *Proceedings of the 2016 international conference on management of data*. 631–646.

[32] Panagiotis Karras and Nikos Mamoulis. 2005. One-pass wavelet synopses for maximum-error metrics. In *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 421–432.

[33] Martin Kiefer, Ilias Poulakis, Sebastian Breß, and Volker Markl. 2020. Scotch: Generating FPGA-Accelerators for Sketching at Line Rate. *Proceedings of the VLDB Endowment* 14, 3 (2020), 281–293.

[34] Taiwo Kolajo, Olawande Daramola, and Ayodele Adebiyi. 2019. Big data stream analysis: a systematic literature review. *Journal of Big Data* 6, 1 (2019), 47.

[35] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. 2006. On-the-fly sharing for streamed aggregation. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 623–634.

[36] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A Tucker. 2005. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *Acm Sigmod Record* 34, 1 (2005), 39–44.

[37] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A Tucker. 2005. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 311–322.

[38] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A Tucker. 2005. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 311–322.

[39] Kaiyu Li and Guoliang Li. 2018. Approximate query processing: What is new and where to go? *Data Science and Engineering* 3, 4 (2018), 379–397.

[40] Charles Masson, Jee E Rim, and Homin K Lee. 2019. DDSketch: a fast and fully-mergeable quantile sketch with relative-error guarantees. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2195–2205.

[41] Frank McSherry, Michael Isard, and Derek G Murray. 2015. Scalability! But at what $COST$?. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*.

[42] Barzan Mozafari, Jags Ramnarayan, Sudhir Menon, Yogesh Mahajan, Soubhik Chakraborty, Hemant Bhanawat, and Kishor Bachhav. 2017. SnappyData: A Unified Cluster for Streaming, Transactions and Interactice Analytics.. In *CIDR*.

[43] Shanmugavelayutham Muthukrishnan. 2005. *Data streams: Algorithms and applications*. Now Publishers Inc.

[44] Gregory Piatetsky-Shapiro and Charles Connell. 1984. Accurate estimation of the number of tuples satisfying a condition. *ACM Sigmod Record* 14, 2 (1984), 256–276.

[45] Viswanath Poosala, Peter J Haas, Yannis E Ioannidis, and Eugene J Shekita. 1996. Improved histograms for selectivity estimation of range predicates. *ACM Sigmod Record* 25, 2 (1996), 294–305.

[46] Do Le Quoc, Ruichuan Chen, Pramod Bhatotia, Christof Fetzer, Volker Hilt, and Thorsten Strufe. 2017. StreamApprox: approximate computing for stream analytics. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. ACM, 185–197.

[47] Madhavapeddi Shreedhar and George Varghese. 1995. Efficient fair queueing using deficit round robin. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*. 231–242.

[48] Apache Spark. 2020. *Scheduling Within an Application*. https://spark.apache.org/docs/latest/job-scheduling.html#scheduling-within-an-application

[49] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. 2003. Load shedding in a data stream manager. In *Proceedings 2003 vldb conference*. Elsevier, 309–320.

[50] NYC Taxi and Limousine Commission (TLC). 2020. New York City Taxi and Limousine Commission (TLC) Trip Record Data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[51] Jonas Traub, Philipp Marian Grulich, Alejandro Rodriguez Cuellar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2018. Scotty: Efficient window aggregation for out-of-order stream processing. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1300–1303.

[52] Jonas Traub, Philipp M. Grulich, Alejandro Rodríguez Cuéllar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2019. Efficient Window Aggregation with General Stream Slicing. In *22th International Conference on Extending Database Technology (EDBT)*.

[53] Jonas Traub, Philipp M. Grulich, Alejandro Rodríguez Cuéllar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2020. Scotty Window Processor. https://doi.org/TU-Berlin-DIMA/scotty-window-processor

[54] Jonas Traub, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2020. Agora: Bringing Together Datasets, Algorithms, Models and More in a Unified Ecosystem [Vision]. *SIGMOD Record* 49, 4 (2020), 6–11.

[55] Jonas Traub, Nikolaas Steenbergen, Philipp M Grulich, Tilmann Rabl, and Volker Markl. 2017. I2: Interactive Real-Time Visualization for Streaming Data. In *EDBT*. 526–529.

[56] Jan E Trost. 1986. Statistically nonrepresentative stratified sampling: A sampling technique for qualitative studies. *Qualitative sociology* 9, 1 (1986), 54–57.

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

[58] Yahoo! 2020. DataSketches: Sketches Library from Yahoo! https://datasketches.github.io/

[59] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.