Synopses for Summarizing Spatial Data Streams

Jacco Kiezebrink, Wieger R. Punter, Odysseas Papapetrou, Kevin Verbeek
Eindhoven University of Technology
Eindhoven, The Netherlands
j.j.e.kiezebrink@student.tue.nl, {w.r.punter, o.papapetrou, k.a.b.verbeek}@tue.nl

***Abstract*—In today's data-driven landscape, geospatial streams are pivotal in diverse fields, ranging from sociology to network engineering and to meteorology. A key challenge in utilizing these streams is to efficiently compute aggregates over ad-hoc spatial ranges, possibly with additional predicates on the stream items. For each application scenario, different aggregates become relevant, such as the number of distinct items (stream arrivals), the count of all items, or even the variance of the frequencies of the items that fall within a spatial range.**

Storing the entire stream for computing these aggregates is impractical in scenarios that involve fast-paced and unbounded streams, due to prohibitive storage costs and query execution delays. To address this, we propose two sketches, SpatialSketch and DynSketch, that support aggregate queries with different types of aggregates. Both sketches require small space, and they can summarize fast-paced streams and estimate the aggregates, with accuracy guarantees. Importantly, both sketches enable supporting new functionalities, in a plug-and-play manner, without requiring novel theoretical analysis. In addition to the theoretical contribution, we evaluate SpatialSketch and DynSketch experimentally. Our experiments demonstrate that the two sketches outperform the state of the art, and that they can be used for addressing novel functionalities for which there exist no small-space solutions to date.

***Index Terms*—Data streams, Synopses, Spatial queries**

I. Introduction

A vast amount of geospatial data is generated on a daily basis from various sources, such as satellites, weather sensors, traffic sensors, smart farming devices [1], and IoT devices [2]. Furthermore, much of these data comes from streaming sources, with high-frequency updates. Indicatively, the National Oceanic & Athmospheric Administration collects around 20 terabytes of environmental data on a daily basis.[1].

The prototypical analytical requirement on such data is for computing aggregate statistics, e.g., counts, sums, and averages, on areas/regions that the user selects at query time. For example, a typical use case in network monitoring includes computing aggregate statistics based on locations/geographical regions, which can reveal DDoS attacks. In traffic monitoring, such statistics are key for quickly detecting traffic abnormalities. In meteorology and climate science, computing aggregate statistics per region is necessary for building weather models [3]. In IoT, exploratory statistics on alarms and events on sensors active in a region at any given moment can be used for testing and auditing procedures [2]. In the mentioned cases, *the regions for which the statistics need to be computed/aggregated are not known a priori* – they are decided, e.g., marked/drawn in a map, at query time, potentially as part of a multi-step analytical pipeline.

Two key challenges arise in the above scenarios. The first challenge involves ingesting the stream. In order to enable answering all future – yet unseen – queries, the whole stream needs to be stored (potentially together with indices). However, the mere data volume generated by fast-paced streams often leads to high storage requirements, which come with a prohibitively high price tag. Furthermore, the storage medium (typically secondary storage, which can store Terabytes of data) may not be able to store all data at line rate, leading to throttling. The second challenge involves efficient query execution. Efficiency is particularly important for fast-paced streams, where the answer changes rapidly. Furthermore, it is imperative in time-critical contexts to minimize querying latency, in order to act fast, e.g., in network monitoring for detecting DDoS attacks before these bring down the corporate network. However, standard techniques and data structures for increasing querying performance, e.g., indexes, are rarely a viable option, as they induce high updating cost and further aggravate storage requirements.

The go-to approach for handling fast-paced streams is with the use of sketches: small-space stream summaries that can be constructed at line rate (typically with constant or poly-logarithmic complexity per update), and can be later used for estimating a variety of queries. Widely used sketches include the Count [4] and Count-min sketch [5] that approximate count queries (how many items with the queried value have been observed in the stream?), the Bloom filter [6] that supports membership queries (has an item been observed in the stream up to now?), and the exponential histogram [7] that summarizes temporal data and allows sliding window aggregate queries (how many items have been observed within a given timeframe). Many other sketches have been proposed in the last years, typically by advancing one of the above key works to support novel requirements.

Our contributions: We describe SpatialSketch, a novel sketch that enables summarization and query estimation over geospatial data. SpatialSketch can also be seen as a sketch wrapper, or a meta-sketch, since it can incorporate existing basic sketches that are not specific to spatial data, in a plug-and-play manner, thereby making them able to answer spatial queries. For example, it can incorporate Count-min sketches for enabling frequency estimates and detection of heavy hitters

This work was partially funded by the European Commission under the STELAR (HORIZON-EUROPE - Grant No. 101070122) project.

[1]https://www.nesdis.noaa.gov/news/data-dive-five-noaa-databases-are-worth-exploring

over spatial ranges, or Bloom filters for enabling membership queries. Importantly, SpatialSketch already comes with the necessary theoretical analysis for supporting a wide range of basic sketches, i.e., it will provide accuracy guarantees out-of-the-box, even when it is instantiated with a new – unforeseen – basic sketch to offer novel functionality. SpatialSketch dictates the following three choices: (a) how basic sketches can be incorporated for allowing novel aggregates on spatial queries, (b) how queries can be executed efficiently, with accuracy guarantees, and (c) how an extension of the sketch, called DynSketch, can dynamically change its resolution, to satisfy memory constraints. Unlike all previous works, SpatialSketch and DynSketch do not limit the queries to rectangular regions – regions of any shape can be handled.

This work advances the state of the art in several aspects. First, compared to exact solutions, SpatialSketch drastically reduces the memory requirements and improves performance, in terms of both data ingestion and query execution. Second, compared to the state of the art approximate algorithm: (a) SpatialSketch is more efficient and more accurate, (b) it supports queries of any shape – not only rectangular/box queries – and, (c) it can utilize different nested sketches with formal probabilistic guarantees, thereby enabling novel functionalities out-of-the-box.

Running example: Before delving into the details, let us briefly discuss a use case that will also be used throughout this document as the running example. A large company wants to be able to analyse the incoming traffic on its networks, i.e., the network packets that enter its network from outside. Such analysis is key for identifying issues in the network, and for optimizing the network connectivity, e.g., by installing additional data centers in *busy* regions. The input is a stream including the IP address of the destination of the network packet (i.e., the one inside the company) and the timestamp of the packet. The stream is augmented with the geographical coordinates of the packet origin, which can be found with the use of any IP geolocation database, looking at the IP address of the sender (Fig. 1). Typical analyses on this stream include finding the number of packets arriving from a region (say, a region selected by the network administrator on a map), or the number of packets that arrive from a region and are also sent to a particular server in the company. The same queries may also be executed on a sliding window basis. Other indicators for detecting DoS attacks include the number of distinct IP addresses from one region that send packets to any one of the company's machines, or even the number of distinct messages originating from a region that communicate with a subset of the company's network.

It is important to note that the spatial ranges, and the potential predicate values are not known a priori in this use case, and that the regions are not necessarily whole cities, provinces, or countries, or even rectangular regions. For example, the region may as well be a university campus, selected from a map, or a neighborhood in a city. These predicates are decided as part of a multi-step data exploration process. Therefore, it is not possible to collect statistics for all possible queries while looking at the stream.

Roadmap: The remaining paper is organized as follows. We will first describe the preliminaries and related work, and explain where the state of the art falls short. In Section III we will describe SpatialSketch, and explain how it can be instantiated. We will show how the sketch is updated and queried, and derive error guarantees for different classes of sketches. In Section IV we will discuss an extension of SpatialSketch, called DynSketch, that works under strict memory quota and is therefore more suitable for use in hardware with limited RAM, such as network routers. Finally, in Section V we will evaluate SpatialSketch with both real-world and synthetic geospatial data, and compare it with the state of the art. Our comparisons will focus on evaluating the efficiency, scalability, and accuracy of each method. We will conclude in Section VI.

II. Preliminaries and related work

Problem definition: This work enables spatial queries on data streams. Consider a (potentially unbounded) data stream $A = \{a_1, a_2, \ldots\}$ of items with the following structure: $< ts, id, [x, y], value >$. Here, ts represents the timestamp and id corresponds to the id of the update, and $[x, y]$ corresponds to the 2-dimensional spatial coordinates of the update. $value$ represents the value of the update (in most cases, value is set to 1). We use $\mathcal{D}$ and $\mathcal{F}$ to denote the 2-dimensional spatial domain and the domain of id respectively.

Spatial queries focus on computing aggregates over spatial ranges, with potentially additional predicates on id. In SQL syntax, the query looks as follows

```
(Query 1) SELECT Agg FROM Stream
WHERE [x,y] in Range
AND (additional predicates)
```

Typical aggregate functions are `COUNT`, `COUNT DISTINCT`, `SUM`, `AVG`, `EXISTS`, and `VAR`, whereas the additional predicates could be, e.g., an equality condition or a range condition on id. The spatial range can be described either as a polygon (e.g., when using spatial SQL extensions), or as a disjunction of conjunctive predicates involving x and y.

As an example, consider the network monitoring scenario of Section I. A query for estimating the number of observed network packets sent by the (almost rectangular) state of Utah is as follows:

```
(Query 2) SELECT COUNT(*) FROM Stream
WHERE -114.052962<x<-109.041058
AND 36.997968<y<42.001567
```

Additional predicates may be interesting, e.g., for only counting the packets that originate from a specific IP address range, and also from Utah:

```
(Query 3) SELECT COUNT(*) FROM Stream
WHERE -114.052962<x<-109.041058
AND 36.997968<y<42.001567
AND int(142.3.1.1) <= id < int(142.7.1.1)
```

with function $int(\cdot)$ denoting the integer representation of the IP address, i.e., reading the IP address as a 4-bytes integer. Alternative predicates may include, e.g., equality predicates on the id or on the value, and predicates on arrival time (sliding window queries).

It is important to note here that *the predicates and the spatial ranges of the query are not known a priori*. In the typical use case, users choose, and possibly progressively revise these predicates, at query time, after part of the stream has been observed.

Dyadic ranges and other techniques for range queries: SpatialSketch relies on dyadic ranges to speed up query execution. Dyadic ranges are all ranges that can be written as $[(x-1)*2^k+1, x*2^k]$, with $x \in \mathbb{Z}^+$, $k \in \mathbb{Z}$. These ranges form a hierarchy, e.g., the dyadic range $[1-32]$ contains two dyadic ranges $[1-16]$ and $[17-32]$, and each of them can be recursively partitioned to smaller dyadic ranges of sizes 8, 4, 2, and 1. Furthermore, any point p from domain $\mathcal{P}$ belongs to exactly $\lceil \log_2(|\mathcal{P}|) \rceil$ dyadic ranges.

A key observation is that any interval $[a, b]$, with $a, b \in \mathcal{P}$, is partitioned to *at most* $2\log_2(|\mathcal{P}|)$ dyadic ranges. Past works, e.g., [5], [8]–[10] exploit this observation to support fast one-dimensional range counts over data streams, as follows. During the monitoring phase, the algorithm maintains a map that keeps the count of items falling in each dyadic range. In particular, for each stream item p, the algorithm uses the map to find all $\log_2(|\mathcal{P}|)$ ranges where p belongs to, and increases their respective counts by 1. At query time, the query range is partitioned to the smallest set of dyadic ranges – at most $2\log_2(|\mathcal{P}|)$ – and the corresponding counts of these ranges are retrieved from the map and summed up to produce the estimate. Alternative implementations that replace the map with arrays, or with a tree, are also possible. The above technique can also be extended to work with two or more dimensions (see [11], Chapter 5.3 for a detailed discussion). Each additional dimension increases the maintenance and querying complexity by a multiplicative factor of $\log(|\mathcal{P}|)$.

An alternative data structure focused on range queries are Fenwick trees [12], which trade off query efficiency for a more compact representation. This is achieved by storing cumulative values from the origin (prefix sums) in a flat array (or a 2-dimensional array if the domain $\mathcal{P}$ is 2-dimensional). Then, query execution amounts to finding the border points of the query, and subtracting the cumulative values before the query range starts from the cumulative value after the query range is completed. This inevitably increases the complexity on query execution by a factor of 2 per dimension. In our experiments (not included in this paper) we found that both Fenwick trees and the dyadic ranges algorithm have very similar performance in practice. Our work could as well use Fenwick trees, albeit with a slightly higher query execution cost.

In the remainder of this section we will describe the key approaches for addressing spatial queries, and explain their key limitations. The approaches will be classified in two groups: (a) exact approaches that store all the data, and, (b) approximate approaches.

A. Related work

Exact algorithms for spatial ranges: Exact algorithms rely on storing the whole stream, e.g., in a relational database or in another binary representation. The stream can then be parsed on demand for answering arbitrary queries. To further speed up query execution, different types of indices can be used – general purpose, such as B-tree [13], as well as specific for spatial data, such as generalized search tree [14], kd-tree [15], and range tree [15]. Most of these indices are implemented in state of the art relational databases, and can be stored both in memory and on secondary storage. However, as we will demonstrate in Section V, these approaches fail in two aspects. First, their space complexity is – at best – linear in the length of the stream, since they need to store and index all data. Second, even though indices are necessary for fast query execution, they slow down the updates, potentially causing load shedding or back-pressure on fast-paced streams.

Approximate algorithms: Approximate algorithms overcome the limitations of exact algorithms by storing summaries/compact representations of the data, albeit with an introduction of a controlled error in the results. Since this work is focused on streams, in the following we will only discuss approximate algorithms that require a single pass over the data. We will not focus on works that require multiple passes, like equi-depth histograms [11] and SliceHist [16], as this is a limiting requirement for streams.

Count-min sketch [5] enables range queries by exploiting dyadic ranges: each arrival is added in the sketch using both its key, and the keys of all dyadic ranges the arrival belongs to. Even though the original paper demonstrates range queries only on one-dimensional domains, it is not difficult to extend the technique to two-dimensional ranges in order to summarize the spatial domain [11]. In this case, the key will then be the coordinates of each stream item, i.e., the geographical coordinates of the IP address. However, this extension can only support count queries; other aggregates like the `L2 norm`, `COUNT DISTINCT`, `EXISTS`, and `COUNT` with additional predicates, cannot be supported, since the information about the arrival – the id in the described representation – is not depicted in the sketch. Furthermore, this extension can only answer rectangular queries. In the spatial domain, most queries have more complex shapes, e.g., a circle, or the shape of a country, a lake, or even a plot of land. The spatial sketch proposed by Das et al. [10] similarly exploits dyadic ranges to index and query the coordinates, but now by using the celebrated AMS sketch [17] for summarizing the input data. However, the sketch comes with limitations similar to the extended Count-min sketch, i.e., it supports only a few aggregate functions (the functionality supported by the underlying AMS sketch), and it can only answer rectangular queries. Our work addresses both constraints.

The state of the art sketch that focuses on higher-dimensional range queries is C-DARQ [18], [19]. For counts over spatial data without additional predicates (e.g., Query 2 above), the two-dimensional spatial domain is segmented

into cells by overlaying a uniformly spaced grid over it. The spatial domain is then separated to cold (sparse) and hot (dense) regions, based on the number of arrivals of the region cells. The cold regions are summarized by a single Count-min sketch, whereas all hot regions are added to a dedicated Count-min sketch, as follows. All cells in a hot region are hierarchically organized in a range tree, with each node in the tree having a unique identifier. For each arrival that belongs to a hot region, the node of the cell in the range tree of its region is found. Then, the unique node id of the node, as well as all node ids of all its ancestors in the tree, are hashed into the dedicated Count-min sketch for the hot regions and their respective counters are increased. For querying, the query range is again partitioned into hot and cold regions. The parts of the query that fall on a cold region are estimated by querying the cold Count-min sketch directly, whereas the hot regions queries are executed by accessing the range trees of these regions top-down. As soon as we find a node of which the range is fully included in the query range, we query the respective Count-min sketch with the unique node id to get an estimate. On the other hand, nodes with zero overlap with the query range are skipped. Finally if the node has partial overlap with the query range, its children are investigated recursively.

The sketch also supports queries with predicates on attributes (e.g., Query 3 above), by considering the attributes as additional dimensions. That is, instead of creating a 2-dimensional sketch (for summarizing only the spatial dimensions), a 3-dimensional sketch can be created, with the third dimension corresponding to the attribute value (in the running example, the IP address). The stream ingestion and the querying algorithms then natively handle the added dimension. A generalization of C-DARQ, called MARQ, supports queries with query ranges that do not co-align with the grid (e.g., the query may include half of a grid cell). It works by keeping samples for each row/column of the grid, and exploiting these to get estimates for the cells that are partially covered. Finally, another extension (DARQ) replaces the two Count-min sketches with Universal sketches [20], to provide support for different types of aggregates. However, all three sketches come with key limitations. First, as we will show in the experiments, their estimates have very high estimation errors for realistically large spatial ranges. In fact, for large spatial ranges that extend beyond one hot region, these sketches need to query the range tree (and the cold region) multiple times, invalidating the theoretical guarantees. Second, similar to the multi-dimensional Count-min sketch, these sketches only support rectangular queries (or, in higher dimensions, box queries). As we will show in Section III, support for non-rectangular queries is not straightforward. Third, even though DARQ is described as being capable of offering different functionalities via the Universal sketch (referred to as different functions f in [19]), the article includes no supporting analysis. In fact, frequently-used functionalities such as distinct count, do not seem to work with DARQ. In our work, we show that the required analysis is not straightforward, and also different analysis is needed for different nested sketch type. We provide this analysis in Section III, and formalize the constraints that the sketches need to satisfy.

TABLE I
FREQUENTLY USED NOTATION

Notation	Description
$\mathcal{D}$	The spatial domain of the stream. All coordinates $[x, y]$ belong in $\mathcal{D}$.
$\mathcal{F}$	The domain of the stream. All ids belong in $\mathcal{F}$.
$x^{\max} \times y^{\max}$	Basic resolution, i.e., resolution of the maximum-resolution grid
g(x, y)	Grid of resolution $x \times y$
R	Arbitrarily shaped range query
r_i	Rectangular sub-range $r_i \in R$
r_i^x	Set of dyadic ranges $\{r_{(i,1)}, r_{(i,2)}, \dots\}$ for the x dimension of rectangular sub-range r_i.
$\mathcal{C}(R)$	Set of dyadic ranges covering range query R.

III. SPATIAL SYNOPSES

We will now describe SpatialSketch, a generic sketch that allows summarization of geospatial streams, and execution of spatial queries of arbitrary shapes. We will first describe the data structure, explain the process of ingesting spatial streams, and discuss its complexity properties. Then we will present the process of estimating aggregates of arbitrarily-shaped queries with SpatialSketch. We will explain how the query range is decomposed into rectangular queries that can be answered efficiently by SpatialSketch, and how SpatialSketch aggregates the individual results of these rectangular queries to produce the final estimate, with error guarantees. We will conclude the section by presenting a few examples of SpatialSketch, for answering different query types. The text will follow the notation and the generic stream model introduced in Section II. As a running example, we will be using the scenario described in Section I.

The data structure: SpatialSketch consists of multiple layers of grids, with each layer summarizing the input stream at a set of different spatial resolutions. The resolution of each grid corresponds to the number of cells of the grid at each dimension. Layer 1 contains a single grid (see Fig. 2, bottom left), which summarizes the input stream at the basic resolution – the maximum supported resolution $x^{\max} \times y^{\max}$ such that the total space required by SpatialSketch does not exceed the memory quota. We will explain later how $x^{\max}$ and $y^{\max}$ are chosen, but these are always powers of two. Layer 2 contains two grids, of resolution $\frac{x^{\max}}{2} \times y^{\max}$ and $x^{\max} \times \frac{y^{\max}}{2}$. In the general case, for each grid of resolution $x_i \times y_i$ contained at layer i, layer $i+1$ contains two grids of resolution $\frac{x_i}{2} \times y_i$ and $x_i \times \frac{y_i}{2}$. This halving process per dimension is interrupted when the number of cells at the dimension reaches 1. Therefore, the top layer contains a single grid of resolution 1×1.

Each cell in each grid contains a basic sketch for compactly summarizing all updates falling in the region covered by the cell. The basic sketch is chosen based on the required querying functionalities. For example, a Count-min or a Count sketch

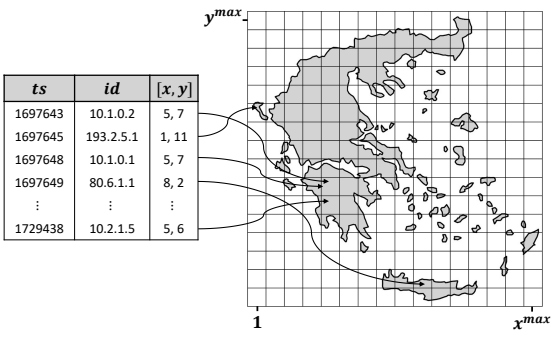

Fig. 1. Example of input stream format with its items mapped to Greece, overlayed by a 16×16 grid.

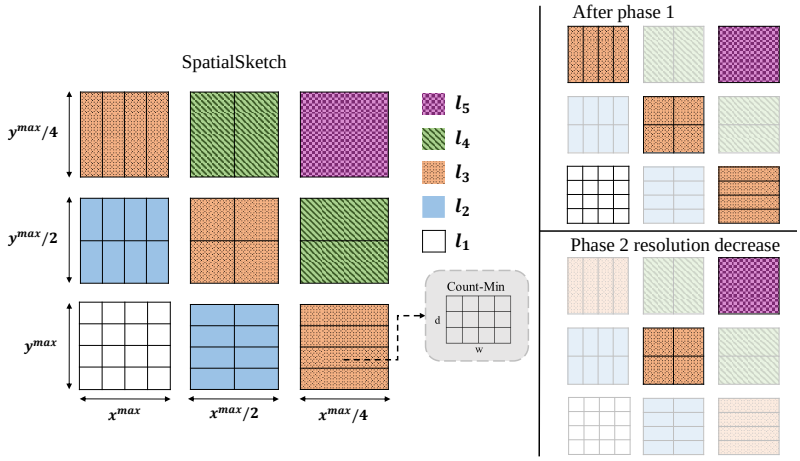

Fig. 2. (Left) SpatialSketch of $x^{\max} = y^{\max} = 4$ containing 9 grids spread over 5 layers. The gray shaded box illustrates the sketch embedded in each cell (Count-Min in this example). (Right) Top-right shows DynSketch after progressively dropping the grids of layers l_2 and l_4 (phase 1 reduction). Bottom-right illustrates a phase 2 reduction, which leads to a reduced resolution.

can be used to allow frequency queries, whereas exponential histograms can be used for sliding window counts.

In the following we will be referring to the individual grids based on their resolution, as: $g(res_x, res_y)$, with res_x and res_y representing the number of grid cells in each dimension. For example, the highest-resolution grid at layer 1 will be labeled as $g(x^{\max}, y^{\max})$ whereas the lowest-resolution grid at the top layer will be labeled as $g(1,1)$. All grids are stored in a hash map G using these labels as keys. Also notice that each grid can be reached by following different halving sequences. For example, grid $g(x^{\max}/2, y^{\max}/2)$ can be reached by first halving dimension x and then halving dimension y, or vice versa. In both cases, the two grids have the same labels, and will end up being identical copies. We avoid creating a grid multiple times by looking whether a grid already exists in the hash map with the same label/resolution.

Space complexity: For computing the number of grids across all layers, notice that the number of cells at dimension x can be $1, 2, 4, 8, \ldots, x^{\max}$ – a total of $\log_2(x^{\max})+1$ values. Similarly, we have a total of $\log_2(y^{\max})+1$ values for the y dimension. By construction, SpatialSketch constructs grids with all possible combinations of these x and y dimensions. Therefore, the total number of grids is $O(\log_2(x^{\max}) * \log_2(y^{\max}))$.

The total number of cells, across all grids, can be computed by summing up the number of cells per grid:

$$\#\text{cells} = \sum_{i=0}^{\log_2(x^{\max})} \sum_{j=0}^{\log_2(y^{\max})} 2^i * 2^j$$
$$< 2^{\log_2(x^{\max})+1} * 2^{\log_2(y^{\max})+1} = 4x^{\max}y^{\max}$$

Since each cell contains a basic sketch, the total space complexity of SpatialSketch is: $O(x^{\max}y^{\max} * B)$, where B denotes the size of the basic sketch. In other words, storing a complete SpatialSketch requires only a multiplicative factor $O(B)$ more than storing a single grid of dimensions $x^{\max} \times y^{\max}$.

Initialization: SpatialSketch is initialized in a lazy manner. At construction time, all grids are created and populate the hash map. The grid cells remain empty, and are initialized with the nested sketch only when an update needs to be stored in that cell. Even though lazy initialization adds up a small overhead at the first stream updates – for allocating memory and initializing the nested sketches – it can lead to significant reductions of the memory requirements, especially when summarizing datasets with skewed spatial distributions.

Updating: Arrivals are tuples of the following[2] format: $< ts, id, [x, y] >$, where ts denotes the timestamp of the arrival, id denotes its id (e.g., the IP address), and $[x, y]$ the coordinates (the location of the IP address). For simplicity, hereafter we assume that all stream arrivals (similarly, all query ranges) have coordinates within the range of the highest-resolution grid, $g(x^{\max}, y^{\max})$: $1 \leq x \leq x^{\max}$ and $1 \leq y \leq y^{\max}$. In practice, this presumes a mapping function map that maps the stream's spatial domain $\mathcal{D}$ to the coordinates of $g(x^{\max}, y^{\max})$. For example, for the standard geographic coordinate system, the 2-dimensional vector function $[x, y] = [\lceil (x' + 90) * x^{\max}/180 \rceil, \lceil (y' + 180) * y^{\max}/360 \rceil]$ can be used to map any $[x', y']$ geographical coordinate to the grid coordinates $[x, y]$. Notice that the mapping function needs not be linear. In fact, SpatialSketch is oblivious to the used spatial domain, coordinate space, and mapping function.

A new arrival $< ts, id, [x, y] >$ is added to SpatialSketch as follows. For each grid $g(i, j)$ of resolution $i \times j$ contained in the hash map G, we add $< ts, id >$ to the basic sketch stored at position $[\lceil x * \frac{i}{x^{\max}} \rceil, \lceil y * \frac{j}{y^{\max}} \rceil]$. The addition method of the chosen basic sketch is integrated without modifications.Each insertion requires updating $O(\log_2(x^{\max}) * \log_2(y^{\max}))$ basic

[2]Additional attributes, such as $value$, can also be contained, and summarized with SpatialSketch, either in place of the id, or as an additional sketch.

sketches – one basic sketch per grid. Therefore, the computational complexity of an insertion grows poly-logarithmically with the basic resolution.

A. Query execution

Queries follow the template of Query 1 (Section II), with `Range` expressed in the coordinates of the highest-resolution grid $g(x^{\max}, y^{\max})$.[3] The query range does not need to be rectangular – it can be of arbitrary shape, e.g., the shape of a campus, or a neighborhood. SpatialSketch also handles queries that have partial overlap with cells by detecting these cells and scaling their estimates proportionally to the overlap, but we cannot provide error guarantees for such queries. Finally, the query can also include additional predicates, e.g., equality predicates on the id, and/or predicates on the arrival time

Query execution process involves two challenges: (a) breaking the arbitrarily-shaped query range to rectangular queries that can be queried efficiently using the constructed grids, and (b) executing the rectangular queries, combining the results, and computing the estimate.

Breaking the query range to rectangular sub-ranges: The naive way to estimate the answer to an arbitrarily shaped query range is to query the contained basic sketches at each of the $g(x^{\max}, y^{\max})$ cells that overlap with the query range, and merge their results. For example, for a count aggregate with Count sketches, all sketches in the query range will be queried and their answers will be summed up. This approach, however, comes with a potentially high computational complexity – linear to the area of the query range.

Instead, SpatialSketch exploits the stored grids of different resolutions to cover the query range efficiently. Each query range R is broken into rectangular sub-ranges $r_1, r_2, \ldots$, which can be further decomposed to range queries for one or more of the stored grids. The decomposition process follows a greedy algorithm, which guarantees near-optimal results [21]. The algorithm starts with a plane sweep [15] for detecting all concave angles within the query range (see transition a in Fig. 3). For each concave angle, we draw one chord (line), parallel to the Y axis. This partitions the query region into a number of smaller, rectangular partitions (transition b). The same process is repeated once more, but now by drawing a chord parallel to the X axis (transition c). Out of the two sets of partitions, we use the one with the smallest number of partitions – in the example, this is the one with the vertical chords. The number of partitions in the chosen partitioning is guaranteed to be at most twice the optimal (minimal) number of partitions for the given query range [21].[4] The

[3]The same function used for the insertion process can be used for mapping the query range to the grid's coordinates.

[4]Multiple heuristics were considered in the literature that were empirically shown to produce partitionings closer to the optimal, with a slight additional cost, e.g., by prioritizing the chords that connect to multiple concave angles. Furthermore, it is also possible to find the partitioning that minimizes the number of partitions [22], yet with much higher computational cost. We have not used them in this work, since query execution is very fast. Still, it is straightforward to use any of these partitioning algorithms in SpatialSketch.

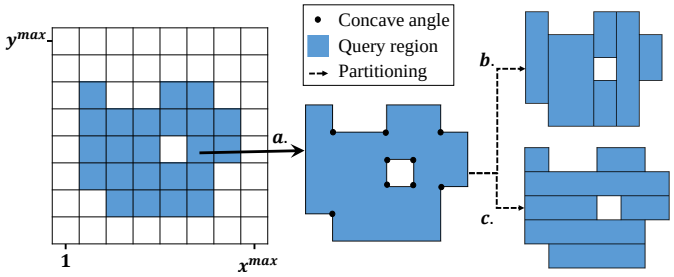

Fig. 3. The process of decomposing a query to rectangular regions: (a) detect all concave angles, (b) draw the vertical chords, which leads to 6 rectangular sub-ranges in this example, (c) draw the horizontal chords, which leads to 7 rectangular sub-ranges.

described algorithm takes $O(V \log V)$ time and space, where V corresponds to the number of angles in the query region.[5]

Querying the rectangular sub-ranges: Up until now, the query has been decomposed into a set of rectangular sub-ranges $r_1, r_2, \ldots$. However, these sub-ranges can be of any size, and there is not necessarily a basic sketch that stores statistics for each sub-range r_i. Therefore, the next step involves breaking each sub-range r_i to dyadic ranges, for which SpatialSketch already contains statistics.

Consider a rectangular sub-range r_i with coordinates $[x_1, y_1, x_2, y_2]$. We begin by breaking down dimension x into its canonical cover. The canonical cover of a range (in this case, the range $[x_1, x_2]$) is the smallest set of dyadic ranges of the form $[(c-1) * 2^k + 1, c * 2^k]$ fully contained within the range, and covering the entire range [10]. Here, c ranges from 1 to $x^{\max}$, and k ranges from 0 to $\log_2(x^{\max})$. To find these dyadic ranges, we start from $k = \lceil \log_2(x_2 - x_1) \rceil$, and find whether there exists a suitable value of c for which $[(c-1) * 2^k + 1, c * 2^k]$ is fully contained in $[x_1, x_2]$. If this is the case, we add $[(c-1) * 2^k + 1, c * 2^k]$ at the canonical cover, and resume the process for covering the remaining ranges $[x_1, (c-1) * 2^k]$ and $[c * 2^k + 1, x_2]$. If this is not the case, we reduce k by one, and try again. The algorithm guarantees that it will take at most $O(\log_2(x^{\max}))$ rounds, and generate at most $O(\log_2(x^{\max}))$ dyadic ranges that fully cover range $[x_1, x_2]$. Let $r_i^x = \{r_{(i,1)}^x, r_{(i,2)}^x, \ldots\}$ denote the set of dyadic ranges for the x dimension of rectangular sub-range r_i. The same process is repeated for the y dimension, yielding $r_i^y = \{r_{(i,1)}^y, r_{(i,2)}^y, \ldots\}$.

Following, we compute the Cartesian product $r_i^x \times r_i^y$, i.e., we combine each dyadic range for x with all dyadic ranges for y. This leads to $O(\log_2^2(x^{\max}))$ 2-dimensional ranges that are guaranteed to be a partitioning of r_i.

Lemma 1. *For any* rectangular range r_i*, the area covered by the union of the ranges contained in the Cartesian product* $r_i^x \times r_i^y$ *is a partitioning of* r_i*, i.e., it contains each cell of* r_i *exactly once, and it does not contain any other cells.*

Let $\mathcal{C}(R)$ denote the set of 2d-dyadic ranges corresponding to a query region R. Notice that, by construction, the

[5]The additional logarithmic factor is caused by the plane-sweep algorithm.

starting and ending coordinates of all ranges contained in $\mathcal{C}(R)$ correspond to dyadic ranges in each dimension. Since the grids in G cover all combinations of 2d-dyadic ranges, for each 2d-dyadic range $[x_1, y_1, x_2, y_2] \in \mathcal{C}(R)$ there exists a grid in G with a cell that summarizes exactly the region $[x_1, y_1, x_2, y_2]$, i.e., each cell in this grid covers a space of size $(x_2 - x_1 + 1) \times (y_2 - y_1 + 1)$. This grid has resolution $\frac{x^{\max}}{x_2 - x_1 + 1} \times \frac{y^{\max}}{y_2 - y_1 + 1}$, and is stored in G with label $g(\frac{x^{\max}}{x_2 - x_1 + 1}, \frac{y^{\max}}{y_2 - y_1 + 1})$.

Lemma 2. *For any 2d-dyadic range $[x_1, y_1, x_2, y_2]$ derived by the described algorithm, there exists a grid $g \in G$ with resolution $\frac{x^{\max}}{x_2 - x_1 + 1} \times \frac{y^{\max}}{y_2 - y_1 + 1}$ whose position $[\frac{x_2}{x_2 - x_1 + 1}, \frac{y_2}{y_2 - y_1 + 1}]$ corresponds to range $[x_1, y_1, x_2, y_2]$.*

Our results up to now guarantee that any arbitrarily-shaped query can be decomposed to set $\mathcal{C}(R)$ of 2-dimensional dyadic ranges, and for each of these dyadic ranges there exists a cell in SpatialSketch covering exactly the same space. The final step involves retrieving the basic sketches stored in these cells, and querying the stored sketches in them to estimate the answer.

To allow for tighter error bounds, we adapt this last step based on the properties/characteristics of the basic sketch. We identify the following key categories of basic sketches:

- Mergeable sketches: a new sketch can be created that summarizes the union of data summarized by the individual sketches, by merging the individual sketches. The merging operation is sketch-dependent.
- Sketches that provide (ϵ, δ) guarantees for distributive queries, with the success probability derived with Markov inequality.
- Sketches that provide (ϵ, δ) guarantees for distributive queries, with the success probability derived with Chebyshev bound.

To the best of our knowledge, these categories cover the majority of available sketches. In the following we discuss each category separately. We will use R to denote the query region and $\mathcal{C}(R) = \{c_1, c_2, \ldots\}$ to denote the cells that comprise the query region (not necessarily of the same size). With $f(q, c_i)$ (resp. $f(q, R)$) we will denote the answer of the query q within cell c_i/query region R, and with $\hat{f}(q, c_i)$ (resp. $\hat{f}(q, R)$) the estimate for query q in the cell/query region.

1) Mergeable sketches: Let $\bigoplus$ denote the merging operation[6] on the basic sketches, and sk_i denote the sketch corresponding to cell c_i. The algorithm first computes $sk_R = sk_1 \bigoplus sk_2 \bigoplus sk_3 \ldots$, and uses it to estimate the query answer. The (ϵ, δ) guarantees of sk_R depend on the merging operation. Most of the existing sketches fall in this category, e.g., the Count-min sketch ($\bigoplus$ is addition), the Bloom filter ($\bigoplus$ is bitwise or), and the ECM sketch (a custom $\bigoplus$ is presented in [23]). The error guarantees are inherited from the sketch's merging operation – SpatialSketch does not affect the error.

[6]Mergeability requires that all sketches are configured with the same parameters – identical hash functions and dimensions. For example, all Bloom filters should be configured with the same k hash functions, and the same length m.

2) Sketches that provide (ϵ, δ) guarantees for distributive queries, with Markov inequality: The queries answered by these sketches satisfy the distributive property, i.e., $f(q, R) = f(q, \bigcup_{c_i \in \mathcal{C}(R)} c_i) = \sum_{c_i \in \mathcal{C}(R)} f(q, c_i)$. In the following we present a generic proof that works for all sketches in this category, but also include the details for the case of Count-min sketch for frequency estimates, as an example.

Each individual sketch sk_i that summarizes the contents of cell c_i offers guarantees of the form $Pr[|\hat{f}(q, c_i) - f(q, c_i)| \leq \epsilon\phi(q, c_i)] \geq 1 - \delta$, where $\phi(q, c_i)$ denotes a function on the cell's contents. The sketch estimation procedure involves executing $d \geq 1$ experiments, and taking the result that minimizes the error. The process on finding the estimate that minimizes the error is specific to the sketch. If each of the experiments provides an $O(\epsilon)$ estimate with constant probability, then, by Markov inequality, the estimate that minimizes the error across all repetitions will guarantee an (ϵ, δ) estimate, as desired.

We adapt the querying process as follows. Let $\hat{f}_j(q, c_i)$ denote the estimate of the j-th repetition, of sketch sk_i, which summarizes cell c_i. We compute $\hat{f}_j(q, R) = \sum_{c_i \in \mathcal{C}(R)} \hat{f}_j(q, c_i)$ per repetition $j = \{1, 2, \ldots, d\}$, and return the estimate of the repetition that minimizes the error, similar to the original sketch.

As an example, consider a SpatialSketch configured with Count-min as the basic sketch. Each Count-min sketch includes $d = O(\log(1/\delta))$ repetitions - rows, and the estimation error at each sketch is bounded by ϵ times the L1 norm, i.e., $\phi(q, c_i)$ equals to the L1 norm of c_i. A query is executed by: (a) retrieving the sketches that correspond to the 2d-dyadic ranges covering the query range, (b) getting one estimate per sketch, per row, (c) summing up the estimates across all sketches per row, and, (d) returning the minimum sum as a final estimate.

Lemma 3. *The estimate of SpatialSketch for a query region R will satisfy $Pr[|\hat{f}(q, R) - f(q, R)| \leq \epsilon \sum_{c_i \in \mathcal{C}(\mathcal{R})} \phi(q, c_i)] \geq 1 - \delta$, when SpatialSketch is instantiated with a basic sketch that provides the following error guarantees $Pr[|\widehat{f}(q, c_i) - f(q, c_i)| \leq \epsilon\phi(q, c_i)] \geq 1 - \delta$ for each individual cell c_i.*

Proof: The proof is included in the technical report [24].

Corollary 1. *For all functions ϕ satisfying $\sum_{c_i \in \mathcal{C}(\mathcal{R})} \phi(q, c_i) \leq \phi(q, R)$, the estimate of SpatialSketch for any query region R will satisfy $Pr[|\hat{f}(q, R) - f(q, R)| \leq \epsilon\phi(q, R)] \geq 1 - \delta$.*

3) Sketches that provide (ϵ, δ) guarantees for distributive queries, with Chebyshev bound: We will present the necessary adaptation and the proof for the Count sketch [4], which is the most widely used sketch in this category. The AMS sketch [17] has similar properties and the proof is similar. As a reminder, the Count sketch is used for estimating frequencies (among with other functionalities). Similar to the Count-min sketch [5], the Count sketch is also a 2-dimensional array, of size $w * d$. It is accompanied by d hash functions $h_j(x)$ that map each input item to $[1 \ldots w]$, and by d sign functions $s_j(x)$, which map the input items to $\{-1, 1\}$, with $\mathbb{E}[s_j(x)] = 0$.

Frequency estimation with the Count sketch is similar to the Count-min sketch, but now including the sign function. In particular, $\hat{f}(q) = \text{median}_{j\in[1\ldots d]} s_j(q) * \text{count}(j, h_j(q))$. To use the count sketch as basic sketch in SpatialSketch we only need to alter the last step of the process described in Section III-A2, such that it returns the median sum per repetition – per row – instead of the minimum sum. So, $\hat{f}(q,R) = \text{median}_{j\in[1\ldots d]} \sum_{c_i\in C(R)} s_{i,j}(q)\text{count}(j, h_{i,j}(q))$, where $h_{i,j}$ and $s_{i,j}$ correspond to the hash function and the sign function of the Count sketch contained at cell c_i, for row j. The following lemma bounds the error of SpatialSketch configured with the Count sketch. The lemma considers the case that all sketches are constructed with different mutually independent hash- and sign functions.

Lemma 4. *The estimate of SpatialSketch for a query region R will satisfy $Pr[|\hat{f}(q,R) - f(q,R)| \leq \epsilon||R||_2] \geq 1-\delta$, when SpatialSketch is instantiated with Count sketches that provide (ϵ,δ) guarantees.*

Proof sketch: There are two key steps in the proof. First, we derive an upper bound on the variance of the estimator:

$$Var\left(\hat{f}_j(q,R)\right) \leq 1/w \sum_{x\in\mathcal{F}\setminus q,\ c_i\in\mathcal{C}(R)} f(x,c_i)^2$$
$$\leq 1/w \sum_{x\in\mathcal{F}\setminus q} f(x,R)^2 \leq ||R||_2^2/w \quad (1)$$

Then, we apply Chebyshev's inequality with the median trick to show that with $d = 8\log(1/\delta)$ and $w = 4/\epsilon^2$ we get $Pr[|\hat{f}(q,R) - f(q,R)| \geq \epsilon||R||_2] \leq \delta$ Formal proof is included in the technical report [24]. □

A practical consideration: The astute reader might have noticed that Inequality 1 is derived by upper bounding $\sum_{c_i\in\mathcal{C}(R)} f(x,c_i)^2$ with $f(x,R)^2$, to get a clearer theoretical result. The resulting bound matches the bound one would get by merging the Count sketch, as described in Section III-A1). If we avoid this simplification we can get a much tighter, but less usable bound for variance. This hints that, by configuring the Count-sketches at each cell with different hash functions and sign functions, we will get smaller errors in practice.

The AMS sketch proof is similar. To the best of our knowledge, these two are the only main sketches in this category.

4) Examples of SpatialSketch with different sketches: The above two classes of sketches cover the majority of sketches from the literature. Table II includes a few frequently used sketches, and explains how these are handled in SpatialSketch. Note that some of the sketches, e.g., exponential histograms, are deterministic. These fall as a special case of the second class of sketches, and the error bounds for these sketches can be derived by setting $\delta = 0$ to Lemma 3 and Corollary 1.

Also notice that some sketches belong to two classes. In these cases, the user may choose the most convenient approach to produce the estimate. For example, ECM-sketch is both mergeable and provides suitable (ϵ,δ) guarantees using the Markov inequality. However, merging of two ECM-sketches slightly increases the value of ϵ (see [23] for the details), whereas the second approach of adding up the individual estimates maintains the original (ϵ,δ) guarantees, and is more computationally efficient. In this case, the second approach is preferred. A similar example is with the Chebyshev bound sketches, where construction of sketches with different hash functions (hence, not mergeable) may lead to better estimates (see our discussion at the end of Section III-A3).

TABLE II
CASE IN PROOF PER BASIC SKETCH. * MERGEABLE SKETCHES INHERIT ALL QUERY TYPES FROM THE BASIC SKETCH.

Basic Sketch	Case (Query Type)
Count-min [5]	Mergeable*, Markov (Frequency)
Count-sketch [4]	Mergeable*, Chebyshev (Frequency)
AMS [17]	Mergeable*, Chebyshev (Frequency)
Bloom filter [6]	Mergeable*
Flajolet-Martin [25]	Mergeable*
HyperLogLog [26]	Mergeable*
Distinct Sampling [27]	Mergeable*
K-minwise hashing [28]	Mergeable*
Universal Sketch [20]	Mergeable*
Misra-Gries [29]	Mergeable* – see [30]
ECM [23]	Mergeable*, Markov (SW Frequency)
Expon. Hist. [31]	Mergeable* – see [23], Markov (SW Count)
Det. Waves [32]	Mergeable* – see [23], Markov (SW Count)
Rand. Waves [32]	Mergeable*

IV. DYNAMIC SPATIALSKETCH

The space requirements of SpatialSketch depend partly on the density and distribution of the input stream. For instance, consider a SpatialSketch configured with a nested Count-min sketch, for summarizing different demographics in Greece (Fig. 1). Many areas of the sketch are sparsely inhabited, being water or tiny islands. Owing to lazy initialization, SpatialSketch will not initialize the nested sketches on the cells that have no input data, drastically reducing memory complexity. This however implies that the memory requirements of SpatialSketch may be significantly overestimated, in the absence of knowledge about the yet-unseen input stream. Another scenario that may lead to either under-estimation or over-estimation of the memory requirements of the sketch is when the space complexity of the nested sketch depends on the frequency of updates in the stream. This is, for example, the case with ECM-sketches [23].

A way to anticipate such cases is to initialize the sketch with a lower basic resolution, leaving plenty of room for the sketch to grow. This, however, may also lead to under-utilization of the available memory, and to larger estimation errors. To avoid this issue, we now propose an extension of SpatialSketch that dynamically adapts to the incoming data, and trades-off query efficiency with memory requirements. The extension, called Dynamic Spatial Sketch (DynSketch for short), relies on two key observations: (a) any query that can be answered by a single cell of a grid at layer $i > 1$ can also be answered by two cells of a grid at layer $i-1$, and, (b) a SpatialSketch of basic resolution $x^{\max}\times y^{\max}$ can be reduced to a SpatialSketch

of basic resolution $\frac{x^{\max}}{2} \times \frac{y^{\max}}{2}$ on-the-fly, without reiterating over the input stream.

DynSketch is initialized with the maximum desired basic resolution and a memory quota, and starts summarizing the input stream. Whenever the sketch needs to grow beyond the memory quota, it enters a two-phase compaction process. In the first phase, some of the stored grids are deleted (layer by layer), without affecting the basic resolution of the sketch. In the second phase, the sketch progressively reduces its basic resolution. The second phase is entered only when the first phase fails to release sufficient memory, i.e., all the grids that could be deleted are already deleted. We will now explain the two phases in more detail.

Phase 1. Let ℓ represent the total number of layers. For example, in the sketch of Fig. 2, which we will be using as a running example, $\ell = 5$ (starting to count from layer 1). When DynSketch runs out of memory, it initially deletes the grids at layer $\ell - (\ell\%2)$, one by one, until it releases sufficient memory. In the example, DynSketch will first delete grid $g(\frac{x^{\max}}{2}, \frac{y^{\max}}{4})$, and, if necessary, grid $g(\frac{x^{\max}}{4}, \frac{y^{\max}}{2})$. [7] The layer ids do not change in this process. When more memory is needed, DynSketch will continue by progressively deleting grids at layer $\ell-2-(\ell\%2)$, $\ell-4-(\ell\%2)$, and so on, until it reaches to layer 1. In the example, after deleting the two grids at layer 4, the sketch will delete the two grids at layer 2.

Phase 2. If DynSketch still needs memory and phase 1 has already completed, i.e., all grids that could be deleted are already deleted, the compaction process enters phase 2. In this phase, the sketch will delete all grids of resolution $x \times y$, where $x = x^{\max}$ *or* $y = y^{\max}$, and subsequently set its basic resolution to $\frac{x^{\max}}{2} \times \frac{y^{\max}}{2}$. In the example of Figure 2, execution of phase 2 will delete the grids forming the faded ⌞-shape. Phase 2 can be repeated every time the sketch outgrows the memory quota. Notice that each execution of phase 2 also reduces the available resolution for selecting the query range, i.e., end-users will be able to choose more coarse-grained query ranges.

Query execution. The first phase of the compaction process requires a small modification at the query execution algorithm of Section III-A. Recall (Lemma 2) that for any 2d-dyadic range $[x_1, y_1, x_2, y_2]$, the querying algorithm expects that there exists a grid g with resolution $\frac{x^{\max}}{x_2-x_1+1} \times \frac{y^{\max}}{y_2-y_1+1}$, which is then used to estimate an answer for the dyadic range. If this grid has already been deleted at phase 1, the dyadic range needs to be further broken down to smaller dyadic ranges, that can be answered by existing (non-deleted) grids.

Observe that phase 1 will only delete grids at alternate layers (in the example, layer 4, and then layer 2, leaving layers 5, 3, and 1 intact). Therefore, if a grid at layer l has been deleted at phase 1, the algorithm breaks the dyadic range $[x_1, y_1, x_2, y_2]$ to two sub-ranges: $[x_1, y_1, x_1 + (x_2 - x_1 + 1)/2 - 1, y_2]$ and $[x_1 + (x_2 - x_1 + 1)/2, y_1, x_2, y_2]$, which can both be answered by an existing grid at layer $l - 1$. This modification increases the query time by at most a factor of two, but does not affect the estimation accuracy, or the theoretical guarantees.

If DynSketch also enters phase 2, the maximum resolution of the sketch is reduced, i.e., the user can now describe the query in a coarser resolution. However, the theoretical guarantees for the chosen query range, as well as the query answering algorithm remain the same.

[7] The order of grid deletion *within a layer* is not important. We attain a deterministic behavior by always deleting the candidate grid with the highest x resolution in the layer.

V. Experimental Evaluation

The purpose of our experiments was twofold: (a) to compare the space complexity, efficiency and accuracy of existing methods to SpatialSketch and DynSketch, and, (b) to evaluate the performance of our sketches in different configurations, and with different stream types and query types. Since DynSketch without a memory quota is reduced to SpatialSketch, in the following we will be using DynSketch for all experiments with memory quota, and SpatialSketch for the rest.

A. Experimental Setup

Hardware and implementation: All experiments were executed on a Linux machine, equipped with 512 GB RAM and an Intel Xeon(R) CPU E5-2697 v2, clocked at 2.7GHz. The experiments were single-threaded, i.e., only one of the 48 cores was used, and the machine was otherwise idle.

Baselines: SpatialSketch and DynSketch were compared with different exact methods (relational database indices from PostgreSQL 16, and spatial indices included in the PostGIS extension), with a sampling approach [33], as well as with the state of the art approximation method, MARQ [18], [19]. C-DARQ [18] was not included in our experiments since it only supports queries that are co-aligned with the grid boundaries, and for such queries, MARQ is reduced to C-DARQ. We also do not consider DARQ (also from [18]), since DARQ does not offer accuracy guarantees for arbitrary sketches (see our discussion in Section II).

Unless otherwise mentioned, SpatialSketch and DynSketch were configured with Count-min basic sketches ($\epsilon = 0.1$, $\delta = 0.05$), and a basic resolution of 4096×4096. Furthermore, for a fair comparison, *all methods (including the PostgreSQL methods) were configured to store all data and auxiliary data structures/indices in RAM*, which resulted to a substantial performance boost for most methods. All methods were implemented in C++. For the MARQ baseline, we started from the GitHub code linked in [18], and implemented the missing functions. All implementations, as well as details for the experiments and datasets are included in our GitHub. [8]

Datasets: We used one real-world dataset from network monitoring – our running example – and synthetically generated datasets. The main dataset was the CAIDA Anonymized Internet Traces dataset [34], which contains passive network traffic traces from the *equinix-nyc* high-speed monitor. Notice that the anonymizing process followed by CAIDA led to invalid (made-up) IP addresses, which could not be mapped to geographical coordinates using an IP Geolocation database.

[8] https://gitlab.com/jaccokiezebrink/grid

To address this, we randomly mapped each anonymized IP address to one existing IP address from the ones contained in the free GeoLite2 dataset [35]. To evaluate the proposed algorithms on different conditions, we also generated a set of synthetic datasets, varying the the geographical distribution of the updates, and the distribution of the attributes. For consistency, all generated datasets follow the same schema with the CAIDA dataset: <ip, x, y>, with ip denoting the target IP address, and x, y denoting the geographical coordinates (latitude and longitude) of the origin of the network packet.

Queries: We considered two query types: (a) rectangular queries, and, (b) polygon queries, with the polygons matching the shapes of US states. The polygon queries were designed by using as query templates the borders of Utah, Nebraska, and Kentucky. These three states were chosen because their borders have varying degrees of complexity: Utah has almost rectangular borders, and is therefore decomposed to a few dyadic ranges, whereas Nebraska and Kentucky require more dyadic range queries. For each query template (both rectangular and polygon), the exact query ranges were generated as follows: (a) the query template was scaled to the desired scale for the experiment, and, (b) the template was randomly placed at a land-covered area in the globe. The reported results always correspond to average numbers, after 100 query executions, i.e., 100 different random query placements. Finally, unless otherwise mentioned, all generated queries had an equality predicate on the IP address, i.e, they followed the template:

```
(Query 4) SELECT COUNT(*) FROM Stream
WHERE [x,y] in Range AND ip=?
```

Since the query template was randomly placed on the map, it was almost always the case that the query range did not co-align with the grid boundaries of SpatialSketch and MARQ. MARQ natively handles this issue by utilizing the samples maintained on each cell (see Section II). SpatialSketch was configured to compute the ratio of the cell that is contained in the query, and use this ratio to scale the statistics coming out of the cell. Finally, note that both SpatialSketch and DynSketch support different aggregate functions. However, the Count aggregate is the only aggregate supported by all compared methods. Therefore, when comparing with the baselines we focused on the count aggregate (similar to Query 3 in Section II). The results for our methods with different aggregates are shown in Section V-D.

Evaluation metrics: For each experiment, we measured: (a) time for ingesting the stream, (b) required RAM, (c) time spent on answering the queries, and (d) approximation error. Unless otherwise noted, approximation error was computed as the absolute difference of the estimate with the exact answer, normalized by the number of updates in the region.

B. Comparison of SpatialSketch to indices that provide exact answers

The goal of the first set of experiments was to compare SpatialSketch to widely-used indices that provide exact answers. The following indices (all available in Postgres) were considered: (a) a B-tree on ip, x, y, (b) a B-tree on x, y, (c) a GiST index on x, y, (d) an SP-GiST index on x, y, (e) no index, i.e., parsing the whole table for each query. For each run, we ensured that the correct index was exploited by looking at the query plan of the query.

Rectangular queries: For the B-tree indices and the no-index case, the query range was expressed as a set of 4 inequality predicates on the coordinates, i.e., $x > x_{\min}$ AND $x < x_{\max}$ AND $y > y_{\min}$ AND $y < y_{\max}$, with $x_{\min}, x_{\max}, y_{\min}, y_{\max}$ denoting the spatial range. The GiST and SP-GiST indices provide a dedicated querying syntax for defining the query range. Fig. 4(a)-(b) plot the ingestion time and memory required by each method, for different stream sizes, whereas Fig. 4(c) plots the query execution time (average over 100 queries) for queries of different spatial ranges. We see that SpatialSketch is at least a factor of 4 times faster and more compact than all exact solutions, including the approach that simply stores all input and does not build an index. Importantly, the memory requirements of SpatialSketch increases in the first few thousand arrivals (not visible in the figure) but then remains constant, whereas all other solutions have a linear space complexity. This makes the exact solutions unsuitable for large streams that run for months, or even years. Focusing on the querying time for each method (Fig. 4(c)), SpatialSketch again outperforms all exact solutions. The B-Tree on ip, x, y is the most efficient of the baseline approaches, but it still requires two orders of magnitude more time compared to SpatialSketch, for large query areas. Since the GiST index is very slow for large query areas, we will no longer consider it.

Polygon queries: The spatial indices offer native support for polygon queries. For the B-Tree indices, polygon queries cannot be handled out-of-the-box. Even though each polygon can be accurately described as a disjunction of conjunctions of functions (e.g., $(3 \leq x < 4$ AND $y - 4x > 0)$ OR $(7 \leq x < 9$ AND $y - 6x > 0)$ OR $\ldots)$, such representation effectively constitutes the B-Trees useless. To enable the use of indices, we applied a grid of 4096x4096 over the whole spatial domain, and then approximated the spatial ranges of all queries by using the cell coordinates. Finally we used the decomposition strategy described in Section III-A to break down the query range to rectangles, which could then be described as SQL predicates. The baseline with no index was handled similarly.

Fig. 4(d) depicts the time required by all methods, as well as the approximation error for SpatialSketch on each querying polygon. The results correspond to queries scaled to cover 10% of the whole space. We see that SpatialSketch consistently outperforms all other methods, and its approximation error is at most 2% – well below the error guarantees of the individual Count-min sketches. We also notice that the only index suitable for polygon queries is the B-tree on ip, x, y. Interestingly, even the spatial indices (GiST and SP-GiST) take two to three orders of magnitude more time compared to SpatialSketch, whereas the B-tree on x, y is not used at all in the query plan, and therefore it makes no difference.

In conclusion, the first round of experiments revealed that

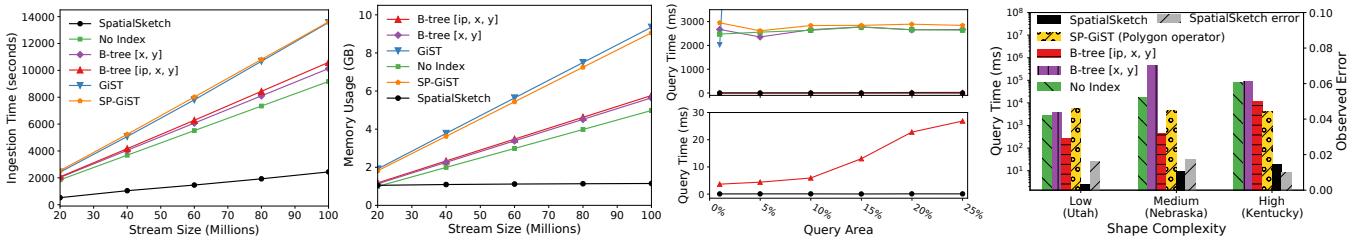

Fig. 4. Comparison with exact solutions: (a) Ingestion time, (b) Memory consumption, (c) Query time for query areas covering between 1% and 25% of the spatial domain (legend identical to plot (b)), and, (d) Query time for non-rectangular queries, covering 10% of the spatial domain, and approximation error of SpatialSketch.

existing indices and exact approaches come with memory requirements that scale linearly to the stream length, and are very slow when executing polygon queries. In contrast, SpatialSketch requires constant space, and estimates even the most complex queries in a few tens of milliseconds.

C. Comparison to approximate methods

The goal for the next set of experiments was to compare DynSketch to the state of the art, MARQ [19], and to reservoir sampling [33]. Our preliminary investigation showed that MARQ has a huge error when one of the predicates is an equality, e.g., ip $=$ 135.6.2.7. In this case, MARQ is effectively reduced to random sampling, with an insufficient number of samples. Therefore, for this experiment, all predicates (including the IP address predicate) are range predicates. The queries follow the template:

```
(Query 5) SELECT COUNT(*) FROM Stream
WHERE [x,y] in Range AND ?<ip<?
```

with randomly chosen spatial ranges and IP address ranges. In order to support range predicates on the IP address, DynSketch was configured with Count-min sketches that support dyadic ranges (see [5] for a description). Since MARQ does not support a fixed memory quota, it was executed first with different ϵ configurations. We then measured its memory consumption and used it as a memory quota for DynSketch and the sampling method.

Fig. 5(a) shows the time spent by each method on ingesting a stream of 100 Million updates, for different memory quotas. We see that DynSketch is slower than the two approximate baselines in terms of ingestion, achieving between 20 and 60 thousand inserts per second. This discrepancy, compared to the previous results of Section V-B where SpatialSketch was achieving much higher insertion rates, is attributed to the basic sketch used by DynSketch now: to enable range queries on the IP address, in this experiment DynSketch uses a Count-min sketch with support for dyadic ranges. Updating this sketch takes $O(\log_2(|\mathcal{F}|))$ more time compared to a simple Count-min – a factor of $32\times$. Reservoir sampling is the fastest approach in terms of stream ingestion, since it requires only $O(1)$ time per update. However, sampling requires between two to three orders of magnitude more time for query execution since it needs to go over the whole sample (Fig. 5(c)). Figure 5(b) plots the approximation error of all methods. Since MARQ is configured for error relative to the total stream length (as opposed to the number of updates falling in the query region, which is supported by DynSketch), for this experiment we normalize the approximation error by the total stream length. We see that MARQ fails to provide reasonably accurate answers. Interestingly, we also observe a stark increase on the error of MARQ when the spatial range of the query is increased, essentially constituting the answers unusable. Comparing our experimental results with the theoretical analysis of MARQ [19], we noticed that the theoretical guarantees of C-DARQ, DARQ, and MARQ, seem to assume that only one hot range tree and only one cold cell needs to be queried per query. In practice, though, large spatial ranges require querying multiple hot range trees and cold cells, leading to a stark increase of the error. Furthermore, the discussed sampling configuration of MARQ fails to maintain a representative sample over the joined coordinates condition.

Summarizing, this series of experiments revealed that DynSketch offers a good balance of accuracy and efficiency, whereas MARQ suffers from high approximation errors, and sampling is slow on query execution. Compared to sampling, for queries with range predicates on the attribute, DynSketch trades off some efficiency on updating for a much faster query execution. For queries with a count aggregate, sampling is also a competitive approach when querying efficiency is not important.

D. DynSketch with different basic sketches

In all previous experiments, DynSketch was configured with a Count-min basic sketch for enabling spatial queries, with a count aggregate. Recall, however, that DynSketch can incorporate other different basic sketches, inheriting their functionality and extending it in the spatial domain. In this series of experiments we used DynSketch for supporting the following functionalities: (a) estimating the self-join size (i.e., the second frequency moment) of the number of network packets sent to each target IP address from within a spatial range, (b) finding whether an IP address received a packet from a spatial range, (c) estimating the number of distinct IP addresses that were sent a network packet from a spatial range.

Table III summarizes the basic sketch used for each functionality, the configured memory, the time required for stream

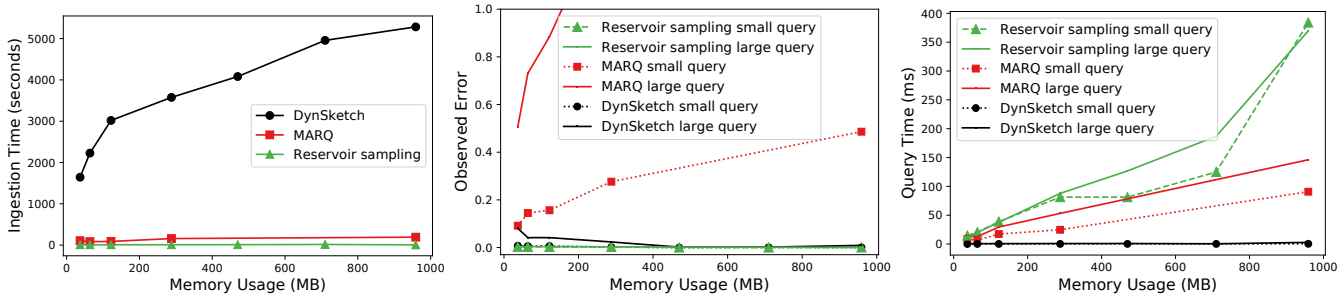

Fig. 5. Comparison with MARQ and reservoir sampling (a) Ingestion time (b) Approximation error (normalized over the stream length) (c) Query time

TABLE III
DYNSKETCH INTEGRATING DIFFERENT BASIC SKETCHES TO ENABLE OTHER FUNCTIONALITIES.

Function	Basic Sketch	Memory	Ingestion	Querying	
				Time	Error
(a) Self-join size	Count-min	37 MB	492 sec	0.019 ms	0.063
(b) Set containment	Bloom filters	710 MB	453 sec	0.0097 ms	3.2% false positives
(c) Count distinct	FM	37 MB	1450 sec	0.25 ms	0.03

TABLE IV
OBSERVED ERROR ON DIFFERENT SYNTHETIC DISTRIBUTIONS FOR SMALL (1%) AND LARGE (25%) QUERIES.

Distr. `ip`	Distr. `x, y`	Error (small)	Error (large)
Uniform	Uniform	0.036	0.036
Uniform	Zipf ($\alpha = 1.3$)	0.085	0.076
Zipf ($\alpha = 1.3$)	Uniform	0.012	0.012
Zipf ($\alpha = 1.3$)	Zipf ($\alpha = 1.3$)	0.028	0.023

ingestion and querying, and the observed error for small rectangular queries. We see that DynSketch supports all three functionalities with low errors. The memory requirements, as well as the ingestion and querying time strongly depend on the basic sketch. For example, Bloom filters require space linear to the number of distinct stream items for a low false positive error, which is also close to the theoretically optimal space complexity for set containment queries [36]. This inevitably leads to large space requirements of DynSketch, as the stream contains more than 1 Million distinct IP addresses. Count-min sketches and FM sketches require much less memory, enabling errors less than 0.05 with less than 37 MB RAM. Similarly, FM sketches are slow, as they require an execution of more than 300 hash functions to theoretically guarantee a 10% relative error. For these three functionalities, the sampling method (not included in the table) provides estimates with very high errors for the same memory quota (for example, 22% false negatives for the set containment queries, and a relative error of 0.58 for counting the distinct items).

In conclusion, this series of experiments confirms the ability of DynSketch to exploit different types of sketches for enabling different functionalities with error guarantees. The compactness of the sketch, as well as the efficiency and error guarantees stem directly from the underlying sketch.

E. Varying the stream properties

At the last set of experiments, we investigated the suitability of DynSketch for summarizing streams of different distributions. We created four synthetic datasets of 100M updates each. For each attribute `x, y, ip`, we chose a characteristic distribution out of uniform, or Zipfian with $\alpha = 1.3$. Then, we generated the stream by randomly drawing values of the chosen distributions. Finally, we generated rectangular queries with equality predicates on the IP address (see Query 4), covering 1% (small) & 25% (large) of the spatial domain. Table IV demonstrates the accuracy of DynSketch with a memory limit of 37MB on the created synthetic streams. We see that DynSketch performed well on all streams, having a maximum error 0.085, which is below the defined $\epsilon = 0.1$ of the corresponding basic Count-min sketch. We highlight that a skewed `x, y` distribution consistently leads to higher errors. On a closer inspection, this error is attributed to cells that are very densely populated, thereby causing a higher overestimation error at the basic sketch (a Count-min, in this case). Still, the error was well below the theoretical maximum of the basic sketches ($\epsilon = 0.1$). The effect of the stream distribution on the ingestion and query time was negligible.

VI. CONCLUSIONS

In this work we considered the challenge of computing aggregate statistics for spatial ranges over data streams. We introduced two novel sketches, SpatialSketch and DynSketch, which can support different types of aggregates (e.g., frequency estimation, L2 norm, membership queries) by incorporating the functionality of other basic sketches like Count-min sketches and Bloom filters. Importantly, the two sketches are backed by a formal analysis that provides accuracy guarantees for diverse aggregates, and for different classes of basic sketches. Through an extensive experimental evaluation with both real and synthetic data streams, we demonstrated that the proposed sketches outperform the competitors (both exact and approximate) in terms of functionality, efficiency, and accuracy, within identical memory constraints.

REFERENCES

[1] H. Bach and W. Mauser, *Sustainable Agriculture and Smart Farming*. Springer International Publishing, 2018, pp. 261–269.

[2] A. Kamilaris and F. O. Ostermann, "Geospatial analysis and the internet of things," *ISPRS International Journal of Geo-Information*, vol. 7, no. 7, 2018.

[3] S. Aisyah, A. A. Simaremare, D. Adytia, I. A. Aditya, and A. Alamsyah, "Exploratory weather data analysis for electricity load forecasting using svm and grnn, case study in Bali, Indonesia," *Energies*, vol. 15, no. 10, 2022.

[4] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," *Theoretical Computer Science*, vol. 312, no. 1, pp. 3–15, 2004, automata, Languages and Programming. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397503004006

[5] G. Cormode and S. Muthukrishnan, "An Improved Data Stream Summary: The Count-Min Sketch and Its Applications," *J. Algorithms*, vol. 55, no. 1, p. 58–75, apr 2005. [Online]. Available: https://doi.org/10.1016/j.jalgor.2003.12.001

[6] A. Broder and M. Mitzenmacher, "Survey: Network Applications of Bloom Filters: A Survey." *Internet Mathematics*, vol. 1, 11 2003.

[7] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," in *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '02. USA: Society for Industrial and Applied Mathematics, 2002, p. 635–644.

[8] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss, "How to summarize the universe: Dynamic maintenance of quantiles," in *Proceedings of the 28th International Conference on Very Large Data Bases*, ser. VLDB '02. VLDB Endowment, 2002, p. 454–465.

[9] W. R. Punter, O. Papapetrou, and M. Garofalakis, "Omnisketch: Efficient multi-dimensional high-velocity stream analytics with arbitrary predicates," *Proc. VLDB Endow.*, vol. 17, no. 3, 2023.

[10] A. Das, J. Gehrke, and M. Riedewald, "Approximation Techniques for Spatial Data," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 695–706. [Online]. Available: https://doi.org/10.1145/1007568.1007646

[11] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine, "Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches," *Found. Trends Databases*, vol. 4, no. 1–3, p. 1–294, jan 2012. [Online]. Available: https://doi.org/10.1561/1900000004

[12] P. M. Fenwick, "A new data structure for cumulative frequency tables," *Software: Practice and Experience*, vol. 24, 1994.

[13] D. Comer, "Ubiquitous b-tree," *ACM Comput. Surv.*, vol. 11, no. 2, p. 121–137, jun 1979. [Online]. Available: https://doi.org/10.1145/356770.356776

[14] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer, "Generalized search trees for database systems," in *Proceedings of the 21th International Conference on Very Large Data Bases*, ser. VLDB '95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, p. 562–573.

[15] M. De Berg, O. Cheong, M. Van Kreveld, and M. Overmars, *Computational Geometry*. Springer Science & Business Media, 3 2008.

[16] M. Shekelyan, A. Dignös, J. Gamper, and M. Garofalakis, "Approximating Multidimensional Range Counts with Maximum Error Guarantees," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, 2021, pp. 1595–1606.

[17] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, ser. STOC '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 20–29. [Online]. Available: https://doi.org/10.1145/237814.237823

[18] R. Friedman and R. Shahout, "Box queries over multi-dimensional streams," in *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 90–101. [Online]. Available: https://doi.org/10.1145/3465480.3466925

[19] ——, "Box queries over multi-dimensional streams," *Information Systems*, vol. 109, p. 102086, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306437922000722

[20] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with UnivMon," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 101–114. [Online]. Available: https://doi.org/10.1145/2934872.2934906

[21] L. Ferrari, P. Sankar, and J. Sklansky, "Minimal rectangular partitions of digitized blobs," *Computer Vision, Graphics, and Image Processing*, vol. 28, no. 1, pp. 58–71, 1984. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0734189X84901397

[22] H. Imai and T. Asano, "Efficient Algorithms for Geometric Graph Search Problems," *SIAM J. Comput.*, vol. 15, pp. 478–494, 1986.

[23] O. Papapetrou, M. N. Garofalakis, and A. Deligiannakis, "Sketching distributed sliding-window data streams," *VLDB J.*, vol. 24, no. 3, pp. 345–368, 2015.

[24] J. Kiezebrink, W. R. Punter, O. Papapetrou, and K. Verbeek, "Synopses for Aggregating Arbitrary Regions in Spatial Data," TU Eindhoven, Tech. Rep., 11 2023. [Online]. Available: https://gitlab.com/jaccokiezebrink/grid/

[25] P. Flajolet and G. Nigel Martin, "Probabilistic counting algorithms for data base applications," *Journal of Computer and System Sciences*, vol. 31, no. 2, pp. 182–209, 1985. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0022000085900418

[26] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier, "HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm," *Discrete Mathematics & Theoretical Computer Science*, vol. DMTCS Proceedings vol. AH,..., 03 2012.

[27] P. B. Gibbons, "Distinct sampling for highly-accurate answers to distinct values queries and event reports," in *VLDB*. Morgan Kaufmann, 2001, pp. 541–550.

[28] R. Pagh, M. Stöckel, and D. P. Woodruff, "Is min-wise hashing optimal for summarizing set intersection?" in *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 109–120. [Online]. Available: https://doi.org/10.1145/2594538.2594554

[29] J. Misra and D. Gries, "Finding repeated elements," *Science of Computer Programming*, vol. 2, no. 2, pp. 143–152, 1982. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0167642382900120

[30] P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, and K. Yi, "Mergeable summaries," *ACM Trans. Database Syst.*, vol. 38, no. 4, dec 2013. [Online]. Available: https://doi.org/10.1145/2500128

[31] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," *SIAM journal on computing*, vol. 31, no. 6, pp. 1794–1813, 2002.

[32] P. B. Gibbons and S. Tirthapura, "Distributed streams algorithms for sliding windows," in *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 63–72. [Online]. Available: https://doi.org/10.1145/564870.564880

[33] J. S. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, vol. 11, no. 1, pp. 37–57, 1985.

[34] "The CAIDA UCSD Anonymized Internet Traces Dataset - [October 2023]," 12 2019. [Online]. Available: https://www.caida.org/catalog/datasets/passive_dataset/

[35] "GeoLite2 Free Geolocation Data - [October 2023]," 10 2023. [Online]. Available: https://dev.maxmind.com/geoip/geolite2-free-geolocation-data

[36] L. Carter, R. W. Floyd, J. Gill, G. Markowsky, and M. N. Wegman, "Exact and approximate membership testers," in *STOC*. ACM, 1978, pp. 59–65.