OmniSketch: Streaming Data Analytics with Arbitrary Predicates

Wieger R. Punter
Eindhoven University of Technology
w.r.punter@tue.nl

Odysseas Papapetrou
Eindhoven University of Technology
o.papapetrou@tue.nl

Minos Garofalakis
ATHENA Research Center & Technical Univ. of Crete
minos@athenarc.gr

ABSTRACT

A key need in different disciplines is to perform analytics over fast-paced data streams, similar in nature to the traditional OLAP analytics in relational databases, i.e., with filters and aggregates. Storing unbounded streams, however, is not a realistic approach due to the high storage requirements, and the delays introduced when storing massive data. Accordingly, many synopses/sketches have been proposed that can summarize the stream in small memory (usually sufficiently small to be stored in RAM), such that aggregate queries can be efficiently approximated, without storing the full stream. However, past synopses predominantly focus on summarizing single-attribute streams, and cannot handle filters and constraints on arbitrary subsets of multiple attributes efficiently. We propose OmniSketch, the first sketch that scales to fast-paced and complex data streams (with many attributes), and supports count aggregates with filters on multiple attributes, dynamically chosen at query time. The sketch offers probabilistic error guarantees, a favorable space/accuracy trade-off, and a worst-case logarithmic complexity for updating and for query execution. We demonstrate experimentally with real and synthetic data that the sketch outperforms the state-of-the-art, and that it can approximate complex ad-hoc queries within the configured accuracy guarantees, with small memory requirements.

1. INTRODUCTION

Filters and aggregates constitute the workhorse of data analytics, and are implemented in all databases. Accordingly, indexing and storage techniques have been implemented to answer such queries efficiently, even over big data. However, analytics on *streaming* data raises new challenges, which call for novel approaches. In particular, due to the large size of contemporary streams – most of them are, in fact, unbounded – storing the full stream such that it can be processed later for answering ad hoc queries becomes impractical. Second, in typical streaming applications, such as monitoring of network traffic data or finance data, extremely fast query response times (in the order of milliseconds) are required to support reactive applications. For example, in network monitoring for early DoS detection, a delay of a couple of seconds when answering a query for up-to-date network statistics may provide sufficient time to an attacker for compromising a corporate network. Third, the high stream update rate (millions of updates per second) that is typical in contemporary streams presents a key obstacle for using traditional data structures to speed-up the queries, e.g., by maintaining different types of indices.

The go-to techniques to handle streaming analytics are *sketches*: compact data structures that summarize the stream, typically with a fixed space complexity. These structures can be updated and queried in constant or polylogarithmic time, with theoretical error guarantees, and they typically work by relying on randomly chosen hash functions to transform the stream into a small-space summary [3]. The key idea is that the user knows the (set of) interesting queries for the user, and initializes the necessary sketches for addressing these queries before observing the stream. Sketches sacrifice some accuracy for a drastic improvement in space complexity, high throughput, and extremely fast querying. Most sketches are also backed by rigorous theoretical analysis, which allow the user to control the space/accuracy trade-off, with probabilistic accuracy guarantees. Owing to their simplicity and high performance, sketches have been widely adopted across distinct domains (both research and industrial applications), to meet diverse requirements. For example, Bloom filters [2] and their variants are frequently used for supporting membership queries on large sets, the celebrated AMS [1] sketches are used for computing L2 norms and similarities over distributions, HyperLogLog sketches [6] are frequently used for estimating distinct counts in streaming multisets, and Count-Min sketches [4] are the de facto data structure for approximating stream distributions.

The key limitation of sketches, however, is that each sketch can be used to address only one, or at best, a few, query templates – predefined structures for aggregate queries on a stream, defined by a *fixed set of predicate attributes* but allowing for variable predicate values. For example, consider the use case of network monitoring, which is crucial for ensuring the reliability, security and performance of network systems. Here, the Count-Min sketch is often used to maintain frequency statistics. A standard IPv4 packet header defines at least 13 attributes, including version, header length, total length, source and destination address, protocol. As a running example throughout this work, we consider a simplified IPv4 header stream of Figure 1, which contains just 4 attributes. To estimate queries adhering to query template `SELECT COUNT(*) FROM STREAM WHERE ipSrc=?` (i.e., number of packets per IP address), a Count-Min sketch using the

 This is a minor revision of the paper entitled OmniSketch: Efficient Multi-Dimensional High-Velocity Stream Analytics with Arbitrary Predicates, published in PVLDB, Vol. 17, No. 3, 2150-8097. DOI: https://doi.org/10.14778/3632093.3632098

ipSrc	ipDest	totalLen	dscp
131.1.2.1	23.11.1.2	40	0
147.3.4.7	147.3.4.8	48	32
147.3.4.7	147.3.4.8	56	32
147.3.4.7	147.3.4.9	56	8
147.3.4.8	147.3.4.7	40	32
...	...	...	...

Figure 1: A (simplified) stream, used as the running example. A sample query with multiple predicates on this stream may be, e.g., `SELECT COUNT(*) FROM stream WHERE ipSrc=131.2.2.1 AND ipDest=142.1.4.7 AND totalLen>40 AND dscp=0`

source IP address as the hash key is required (see Section 2). Monitoring the number of packets received per destination IP (`ipDest`) can be done with another query template – thus an additional sketch that hashes on `ipDest`. To monitor the number of packets exchanged between any two IPs, we need to maintain yet another sketch, that uses the concatenation of the source-destination IP addresses as key.

In exploratory data analysis, the analyst requires the flexibility to answer *any* frequency query on the IPv4 stream, *on any possible combination of predicates*. However, the number of Count-Min sketches to support any potential query template grows exponentially with the number of attributes. In the toy example of Fig. 1, already $(2^4 - 1)$ sketches will need to be maintained, whereas in the non-simplified scenario with all 13 attributes, we would need to maintain $O(2^{13})$ sketches. This requirement stems from the way the sketches are constructed and queried: inserting a tuple in the sketch is done by hashing together the same subset of attributes that will be used as query predicates later. Generalizing to arbitrary scenarios, estimating frequency distribution for all possible combinations of p predicates requires the maintenance of $2^p - 1$ sketches. This is clearly infeasible, because of both the involved space complexity and the strict efficiency constraints that arise in the context of data streams.

Our contribution: OmniSketch.

OmniSketch is the first sketch that allows compact and efficient summarization of (possibly unbounded) fast-paced streams, and subsequent efficient execution of general queries, decided at query time, with accuracy guarantees. The sketch combines the compactness of sketches, which is necessary for reducing the memory constraints, with the generality of sampling, which is key for supporting general queries, on predicates that are dynamically decided at query time. In a nutshell, OmniSketch consists of one compact sketch per attribute – a 2d matrix, similar to Count-Min sketch. Each cell in these matrices contains a fixed-size sample of the inserted record-ids that hash in the cell. At query time, abritrary conjunctions of predicates are supported by looking-up each predicate individually on the corresponding attribute sketch, and computing the intersection of the results (the retrieved record-ids), thereby translating the conjunction of predicates into a set-intersection problem.

Unlike previous work, OmniSketch offers computational complexity (for both updates and queries) that scales *linearly* with the number of attributes – instead of exponentially – rendering it *the only viable, general-purpose solution, to date, for summarizing fast-paced streams with many attributes in small space and with efficient querying.* The sketch comes with formal error guarantees, and an automated initialization algorithm that builds on the theoretical analysis to fully utilize the available memory.

We evaluate OmniSketch experimentally on both real and synthetically-generated streams, and compare it with Hydra, the state-of-the-art competitor. Our experiments confirm that OmniSketch is the only viable option for summarizing complex streams, and comes with a favorable accuracy/efficiency trade-off. In contrast, Hydra becomes extremely slow when summarizing streams with five or more attributes, and therefore fails to effectively summarize fast-paced streams.

Roadmap.

The remainder of the paper is structured as follows. In Section 2 we present the preliminaries and discuss the related work. In Section 3 we present OmniSketch and analyse its theoretical properties, whereas Section 4 summarizes our experimental results. We summarize the work and conclude with future plans in Section 6.

2. PRELIMINARIES

OmniSketch inherits the basic structure of Count-Min sketches [4], and builds on the theory of min-wise hashing [18]. We will now briefly present these two works, to the depth required for understanding our work.

Count-Min sketch.

The Count-Min sketch, proposed in 2005 [4], quickly became the de facto sketch for approximate frequency distributions of streaming data. A Count-Min sketch CM is a 2-dimensional array of width w and depth d, accompanied by d pairwise-independent hash functions $\{h_1, h_2, \ldots, h_d\}$ that map the input to the range $\{1, \ldots w\}$. Let $CM[j,k]$ denote the counter at cell (j,k) in the sketch's 2-dimensional array. A record r (e.g., an IP address) is added once to the sketch by increasing the count at $CM[j, h_j(r)]$, for all $j \in \{1 \ldots d\}$. The frequency of any query item q in the stream can be estimated by finding the corresponding cells $CM[j, h_j(q)]$ per row j, and returning the minimum count, i.e., $\hat{f}(q) = \min_{j \in \{1 \ldots d\}}(CM[j, h_j(q)])$. Due to hash collisions (items other than q that fall in the same counters) $\hat{f}(q)$ is potentially an overestimate of the true frequency $f(q)$. By setting $w = \lceil e/\epsilon \rceil$ and $d = \lceil \ln(1/\delta) \rceil$, we have that $\hat{f}(q) - f(q) \leq \epsilon N$ with probability $\geq 1 - \delta$, where N is the length of the stream.

It is straightforward to use Count-Min sketches for summarizing frequency distributions of pairs, triples, or arbitrary subsets of attributes. For instance, in the running example of Fig. 1, one could summarize the distribution of `<ipSrc,ipDest>` by simply using the concatenation of the two attributes as a *composite* key in all hashes – both at insertion and querying. A single Count-Min sketch is then required, per attribute combination. This approach however becomes prohibitively expensive when we need to summarize many such combinations. Even worse, in the typi-

cal exploratory data analytics scenario where the interesting combinations are decided at query time – based on the intermediary results – the number of sketches that need to be constructed grows exponentially to the number of attributes. Hence, this naive approach does not scale.

Min-wise hashing.

OmniSketch relies on min-wise hashing to estimate the cardinality of the intersection of multiple sets. The key idea behind all min-wise hashing schemes is to hash all items in each set using one or more hash functions, and to keep only the B items with the smallest hash values per hash function as samples of each set. The size of the intersection of these samples for the different sets can then be scaled to estimate the cardinality of the intersection of the sets.

The min-wise hashing algorithm that we will be using in this work estimates the sets intersection cardinality as follows [18]: Let $R_1, R_2, \ldots, R_p$ denote the p sets for which we want to estimate the cardinality of their intersection. First, we construct the summary S_i for each set R_i by hashing each item with a global hash function $g(\cdot) \rightarrow \{0,1\}^b$, and retaining the B smallest hash values, with b set to at least $\lceil \log(4B^{2.5}/\delta) \rceil$. The cardinality of the intersection of p sets $|R_\cap| = |R_1 \cap \ldots R_p|$ can then be estimated from these summaries as $\hat{R}_\cap = |\bigcap_{i=1}^{p} S_i| * n_{\max}/B$ with $n_{\max} = \max_{i=1}^{p} |R_i|$. This estimate comes with (ϵ, δ) guarantees when $\hat{R}_\cap \geq 3n_{\max}\log(2p/\delta)/(B\epsilon^2)$. When the above condition fails, a weaker bound can be shown: $0 \leq |R_\cap| \leq 4n_{\max}\log(2p/\delta)/(B\epsilon^2)$, with probability $1 - \delta\sqrt{B}$.

We chose min-wise hashing over other sampling methods, since this algorithm has been shown to perform equally well or outperform other sampling methods (including [13, 14]), and has space complexity that nearly matches the theoretical lower bound for the problem [18]. Furthermore, the chosen algorithm works with streaming data, as the samples can be maintained incrementally.

The stream model and supported queries.

The input data is a stream of records, e.g., similar to the records of our running example of Figure 1. Let $\mathcal{A} = \{a_1, a_2, \ldots, a_{|\mathcal{A}|}\}$ denote the stream attributes (in the running example, ipSrc, ipDest, totalLen, dscp). Each record also comes with a unique record id (rid), denoted by a_0. If no record keys exist, such unique rids can be easily constructed at ingestion time (e.g., using an arrival counter). We assume a landmark stream query model [17], where a query can be posed any time, and refers to all stream arrivals until that time. Formally, let $\mathcal{R} = \{rec_1, rec_2, \ldots rec_N\}$ denote the stream arrivals up until query time. The query q is a count query, comprising a conjunction of selection predicates for *any* subset $\{a_i, a_j, a_k, \ldots\}$ of $\mathcal{A}$:

$$\text{Count}\,(rec \in \mathcal{R} \mid rec \text{ satisfies } q = q_i \wedge q_j \wedge q_k \ldots)$$

where each predicate q_i can be a disjunction of range and equality predicates on an individual attribute a_i.

Other Related Work.

The state-of-the-art sub-population sketch for multi-dimensional data streams is Hydra [16]. At the outer layer, Hydra consists of a Count-Min sketch of width w and depth d, accompanied with d pairwise-independent hash functions that map the input domain to $[1 \ldots w]$. Each cell in this sketch contains a nested universal sketch [15], for keeping detailed statistics. When a record of p attributes is read, it is hashed as follows. First, all $2^p - 1$ possible combinations of the record's attributes are computed. Each of these combinations defines a sub-population where this record belongs. Each combination is then hashed with the d hash functions to find the corresponding cells in the outer sketch (one cell per row), and the combination is finally added in the contained universal sketches. For query execution, the query predicates are combined to create a key, which is then hashed using the same d hash functions for finding the corresponding cells in the outer sketch. We then query the nested universal sketch using the same key, to estimate the frequency.

In a thorough experimental evaluation, Hydra is shown to outperform other approaches, and provides interactive query latency. However, the sketch has two critical downsides that make it a non-viable option for streams with many attributes. First, recall that each record is added $2^p - 1$ times in the sketch, with different keys – once per possible combination. As we will demonstrate later, this exponential complexity becomes a challenge already for a modest number of indexed attributes, i.e., $p = 4$, in terms of time complexity, causing problems when summarizing fast-paced streams. Second, because of this sharp increase (exponential with p) of the number of additions to the sketch, the approximation error also rapidly increases. Our experimental results show that while Hydra demonstrates excellent performance for small p values, its error (as well as update time) increase exponentially with p.

Techniques for set cardinality estimation.

A number of other sampling-based techniques have been proposed in the literature for estimating the cardinalities of set unions, set intersections, and arbitrary set expressions over (insert-only) record streams [8, 5, 13, 11, 12]. At their core, these methods are similar to min-wise hashing, with similar complexities and theoretical guarantees. In principle, any of these could also be used in OmniSketch to construct the nested sketch. It is important to note, however, that none of these methods (including min-wise hashing) provides support for predicate-based filtering. Consequently, they cannot be directly utilized to estimate answers to aggregate queries with predicates, which is the main focus of our work.

3. OMNISKETCH: FREQUENCY ESTIMATION WITH ARBITRARY PREDICATES

We now present OmniSketch, a sketch that allows efficient frequency estimation of queries with predicates over fast-paced streams. We gradually construct the final sketch, in three successive steps. Initially, we describe an extension of the standard Count-Min sketch, termed $\mathbb{S}0$, that maintains additional data in the cells. This additional information is leveraged for estimating cardinalities involving predicates using a straightforward generalization of the Count-Min estimator. Then, keeping the $\mathbb{S}0$ sketch structure unchanged, we present an improved estimation algorithm and the corresponding theoretical analysis to tighten the error bounds, at no extra complexity. We refer to the new sketch estimator as $\mathbb{S}0_\cap$, to distinguish from the earlier estimator (termed $\mathbb{S}0_{\min}$). Finally, we present the ultimate OmniSketch ($\mathbb{S}1$)

Table 1: Frequently used notation.

Notation	Description
N	Stream length
f(q)	Number of records satisfying query q
d	Sketch depth (number of rows)
w	Sketch width (number of columns)
B	Maximum sample size per cell
CM_i	Sketch for attribute i
$CM_i[j,k]$	Cell at row j, column k, of sketch CM_i
$h_i^j(\cdot)$	Hash function for attribute sketch, row j of attribute i
$S_i(j,k)$	Set of records in the sample of cell $CM_i[j,k]$
$C(q)$	Set of cells $CM_i[j,h^j(v_{q_i})]$ accessed to answer the query
$n_{\max}$	Max. number of records hashed to any cell $CM_i[j,k] \in C(q)$

which combines $\mathbb{S}0_\cap$ with a sampling technique to guarantee sublinear space complexity. Unlike previous works, all three sketches allow very efficient updates *and* queries, with logarithmic complexity.

All three sketches are based on a common architecture: each sketch comprises a collection of sub-sketches, with one sketch assigned to each searchable attribute, i.e., an attribute for which predicate support is desired. We refer to the sub-sketches as attribute sketches, and denote them by CM_1, CM_2, ... $CM_{|\mathcal{A}|}$, where $\mathcal{A}$ corresponds to the set of searchable attributes. Each CM_i is an array of size $w \times d$, accompanied with d pairwise-independent hash functions $h_i^1(\cdot), h_i^2(\cdot), \ldots, h_i^d(\cdot)$ (similar to Count-Min sketches). However, unlike traditional CM-sketches, each cell in CM_i contains a list of record ids (rids), or their fingerprints. Rids are placed/hashed in each sketch based on the record value on the corresponding attribute. During query execution, the attribute sketches corresponding to the query's predicates are queried, and the record ids retrieved by these are intersected, to compute an estimate of the answer. Figure 2 illustrates the general sketch architecture, for a sample dataset with searchable attributes: <ipSrc, ipDest, totalLen, dscp>. The difference between the three sketches relates to: (a) the way the record ids are sampled and stored in the attributes sketches, and, (b) the theory backing the estimators, which affects the tightness of the bounds.

In the ensuing discussion, we use $\mathcal{A} = \{a_1, a_2, \ldots, a_{|\mathcal{A}|}\}$ to denote the set of searchable attributes, q to denote a query that contains p predicates, and q_i to denote the predicate for attribute $a_i \in \mathcal{A}$. For ease of presentation, unless otherwise mentioned, we assume that each predicate q_i is an equality predicate on attribute a_i with a constant value v_{q_i}; that is, $q_i := (a_i = v_{q_i})$. The domain of each attribute a_i is denoted by $\mathcal{D}(a_i)$. Without loss of generality, we assume that q contains all attributes of $\mathcal{A}$, i.e., $p = |\mathcal{A}|$ (if some of the attributes are not contained in the query, we simply ignore the corresponding sketches during query execution); in general, $p \leq |\mathcal{A}|$. The notation is summarized in Table 1.

3.1 Sketch $\mathbb{S}0$: Count-Min with rid-sets for estimating queries with predicates

Initialization.

At initialization time, we choose uniformly at random $|\mathcal{A}| \times d$ pairwise-independent hash functions $h_1^1(\cdot)$, $h_2^1(\cdot)$, $\ldots, h_{|\mathcal{A}|}^d(\cdot)$, with $h_i^j() : \mathcal{D}(a_i) \to \{1, \ldots, w\}$. We also construct $|\mathcal{A}|$ arrays $CM_1, CM_2 \ldots, CM_{|\mathcal{A}|}$, of size $w \times d$, and initialize each of their cells to contain an empty linked list. In order to get (ϵ, δ)-guarantees on the estimate, we set $w = 1 + \lceil e/\epsilon \rceil = \Theta(1/\epsilon)$ and $d = \lceil \ln(1/\delta) \rceil$.

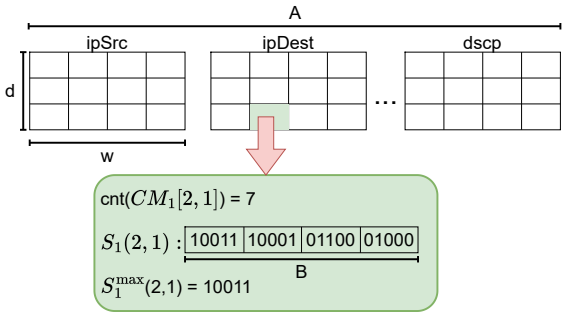

Figure 2: OmniSketch. The green-shaded box illustrates the contents of a cell in $\mathbb{S}1$.

Insertion.

A new record r needs to be added to all $|\mathcal{A}|$ sketches. We use r_i to denote the value of record r on attribute a_i, and r_0 to denote its unique record id (rid). For each searchable attribute a_i and for each row $j = \{1, \ldots, d\}$, we add r_0 to all linked lists at positions $CM_i[j, h_i^j(r_i)]$.

Note that, as defined, the $\mathbb{S}0$ structure is not strictly a "sketch" since, by storing complete rid-sets, it requires space that is linear in the stream length N. The only summarization $\mathbb{S}0$ performs is by "blurring" individual attribute values through hashing into Count-Min buckets. Still, it provides a conceptually useful first step towards our final OmniSketch solution.

Query Estimation.

The $\mathbb{S}0_{\min}$ Estimator. The $\mathbb{S}0_{\min}$ estimator follows the conventional Count-Min estimation procedure [4]. It estimates the count $f(q)$ of records satisfying all predicates in q by computing a per-row intersection of all records stores in the attribute sketches of the predicates in q, and returning the minimum of the per-row estimates. The estimator provides probabilistic guarantees (Lemma 3.1 in [19]) with error bounds proportional to ϵN.[1]

The $\mathbb{S}0_\cap$ Estimator. Similar to standard Count-Min estimation, it is easy to see that $\mathbb{S}0_{\min}$ can only overestimate the true count due to hash bucket collisions (false positives). However, compared to the standard Count-Min, each bucket in $\mathbb{S}0$ contains much more detailed information that can be exploited to provide much tighter estimates (i.e., with less false positives). The key observation here is that, by construction of $\mathbb{S}0$, each record that satisfies the full query will end up in the cells of *all* d rows for this query and for all p predicates, whereas false positives are expected to appear only in a few of the rows. Thus, without modifying the sketch construction or space/time complexity, we can get a tighter estimator by replacing the min operation across the rows with an intersection operation of all the rid-sets returned by all d rows. We refer to this tighter $\mathbb{S}0$ estimator as $\mathbb{S}0_\cap$, and formally define it as follows:

$$\hat{f}(q) = \left| \bigcap_{i=\{1\ldots p\}, j=\{1\ldots d\}} CM_i[j, h_i^j(v_{q_i})] \right| \quad (1)$$

Clearly, this new $\mathbb{S}0_\cap$ estimator can also only overestimate

[1] More details and all proofs are included in [19].

the true count due to hash collisions – at the same time, it guarantees fewer false positives since it is always less than or equal to the $\mathbb{S}0_{\min}$ estimate. The following lemma provides probabilistic guarantees for the $\mathbb{S}0_{\cap}$ estimator.

LEMMA 1. *Let $\hat{f}(q)$ be the estimate provided by Eqn. 1 on sketches constructed with $d = \lceil \ln(\frac{1}{\delta}) \rceil$, $w = 1 + \lceil \frac{e}{\epsilon} \rceil$. Then:*

$$Pr\left(|\hat{f}(q) - f(q)| \leq \epsilon^d (N - f(q)) \leq \epsilon^d N\right) \geq 1 - \delta$$

3.2 Sketch $\mathbb{S}1$ (OmniSketch): Achieving sublinear space through sampling

Sketch $\mathbb{S}0$ stores the record ids of all records, resulting in a space complexity of $O(d \times |\mathcal{A}| \times N)$, which is not viable for large data streams. Our full-fledged OmniSketch (also denoted by $\mathbb{S}1$ in what follows), achieves space complexity strictly sublinear in N by taking and maintaining *samples* of rids in the cells of the attribute sketches, using a min-wise hashing algorithm [18]. We now describe this solution, assuming that two key parameters (the maximum sample size per cell B and the range of the sampling hash function $[0, 2^b - 1]$) are already set. We will explain later in this section how these values are determined.

Similar to $\mathbb{S}0$, $\mathbb{S}1$ is composed of a collection of $|\mathcal{A}|$ attribute sketches. Furthermore, $\mathbb{S}1$ incorporates a hash function $g : \mathcal{D}(r_0) \rightarrow \{0,1\}^b$, which maps each rid to a bit string of length b, with b set to at least $\lceil \log(4B^{2.5}/\delta) \rceil$. Function g is necessary for min-wise hashing (Section 2). The $|\mathcal{A}|$ attribute sketches $CM_1, CM_2, \ldots, CM_{|\mathcal{A}|}$ are all of the same size $w \times d$. Each sketch cell $CM_i[j,k]$ contains: (a) the count of all items that were hashed in this cell, denoted as $cnt(CM_i[j,k])$, (b) a min-wise sample $S_i(j,k)$ of maximum size B, of the rid hash values $g(r_0)$ of all records r that were hashed in this cell, and, (c) the maximum hash value of all items contained in $S_i(j,k)$, denoted as $S_i^{\max}(j,k)$. Notice that $cnt(CM_i[j,k])$ also counts for items that were hashed in $CM_i[j,k]$ but did not end up in the sample, and it is expected to be much greater than B in a populated sketch. Fig. 2 (the green-shaded box) shows an example of a populated cell's contents, with $B = 4$ and $b = 5$.

Initialization.

The user chooses the desired error guarantees (ϵ, δ), with $\epsilon < 0.25$. Let (ϵ_1, δ_1) and (ϵ_2, δ_2) correspond to the (ϵ, δ) configurations of the min-wise sampling algorithm and the attribute sketch respectively, and $\epsilon_1 = \epsilon$, $\epsilon_2 = (\epsilon/(1+\epsilon))^{1/d}$, and $\delta_1 = \delta_2 = \delta/2$. We initialize all attribute sketches, by choosing w and d similar to $\mathbb{S}0$. Each cell in these sketches is initialized with an empty sample $S_i(j,k)$, and with $cnt(CM_i[j,k]) = 0$ and $S_i^{\max}(j,k) = \infty$. Next to the attribute sketches, we initialize one random hash function g, that will hash rids to signatures.

Insertion.

To add a new record r to the sketch, we first locate all cells across the $|\mathcal{A}|$ sketches that correspond to the values of r (lines 2-4, Alg. 1). These are the cells $C(r) = \{CM_i[j, h_i^j(r_i)] : i = \{1, \ldots, |\mathcal{A}|\}, j = \{1, \ldots, d\}\}$. For each of these cells $CM_i[j,k] \in C(r)$, we increase $cnt(CM_i[j,k])$ by one (line 5). Then, we examine whether $g(r_0)$ should be added to the sample of the cell (lines 6-10), as follows. If the sample contains less than B items, then $g(r_0)$ is added to it. If, on the other hand, $|S_i(j,k)| = B$ and $g(r_0) < S_i^{\max}(j,k)$ (i.e., the sample is full, but the new record's id has a smaller hash value g compared to another record from the sample), we remove $S_i^{\max}(j,k)$ from the sample to make space for $g(r_0)$. Finally, we add $g(r_0)$ to $S_i(j,k)$, and recompute $S_i^{\max}(j,k)$. [2] The complexity of inserting an element is $O(|\mathcal{A}| \times d \times \log(B))$.

Algorithm 1 Adding a record r to the sketch

```
1:  Input: record r
2:  for i ∈ {1 ... |A|} do
3:      for j ∈ {1 ... d} do
4:          k = h_i^j(r_i)
5:          cnt(CM_i[j,k]) ++
6:          if |S_i(j,k)| < B then
7:              Add g(r_0) to S_i(j,k)
8:              Recompute S_i^max(j,k)
9:          if |S_i(j,k)| == B then
10:             if g(r_0) < S_i^max(j,k) then
11:                 Remove S_i^max(j,k) from S_i(j,k)
12:                 Add g(r_0) to S_i(j,k)
13:                 Recompute S_i^max(j,k)
```

Query Estimation.

Let q denote the input query. As with $\mathbb{S}0$, we assume that q contains all $\mathcal{A}$ attributes, i.e., $p = |\mathcal{A}|$; if an attribute is not contained in q, the estimator simply ignores the corresponding attribute sketch. Each query attribute value v_{q_i} is hashed using the d hash functions of the corresponding sketch CM_i, which leads to d cells per sketch. Let $C(q)$ denote the set of cells across all sketches that are accessed to answer the query, i.e., $C(q) = \{CM_i[j, h^j(v_{q_i})] : i = \{1 \ldots p\}, j = \{1 \ldots d\}\}$, and $n_{\max}$ be the maximum count $cnt(CM_i[j,k])$ of these cells. Let

$$S_{\cap} = \bigcap_{i=\{1\ldots p\}, j=\{1\ldots d\}} S_i(j, h^j(v_{q_i}))$$

denote the intersection of all samples stored in all cells in $C(q)$. Our OmniSketch estimator $\hat{f}(q)$ is computed as:

$$\hat{f}(q) = \frac{n_{\max}}{B} \times |S_{\cap}| \tag{2}$$

Intuitively, this estimator computes the cardinality of the intersection of all samples in $C(q)$ and scales it up by a factor of $\frac{n_{\max}}{B}$, to account for the sampling. Based on the analysis of min-wise hashing [18], our estimator comes with a sanity bound of $\frac{3 n_{\max} \log(4pd\sqrt{B}/\delta)}{B\epsilon^2}$ to cover the case of insufficient samples in the intersection.

Derivation of the error bounds.

The following theorem provides the error guarantees for our OmniSketch estimator. The nature of the guarantee – whether it depends on ϵN or on $n_{\max}/\epsilon^2$ – is determined by the number of witnesses of the intersection ($|S_{\cap}|$). It is important to note that in any realistic setting, $n_{\max} << N$.

[2] To speed-up the computation of $S_i^{\max}(j,k)$, as well as the estimation process, the samples $S_i(j,k)$ are maintained in a red-black tree. Therefore, re-computation of $S_i^{\max}(j,k)$ takes (amortized) constant time.

THEOREM 2. *Consider an OmniSketch with $\epsilon_1 = \epsilon$, $\epsilon_2 = \big(\epsilon/(1+\epsilon)\big)^{1/d}$, and $\delta_1 = \delta_2 = \delta/2$. Let $\hat{f}(q) = \frac{n_{\max}}{B} \times |S_\cap|$. If $|S_\cap| < \frac{3n_{\max}\log(4pd\sqrt{B}/\delta)}{B\epsilon^2}$, then $|f(q) - \hat{f}(q)| \leq \frac{4n_{\max}\log(4pd\sqrt{B}/\delta)}{B\epsilon^2}$ with probability at least $1-\delta/2$. Otherwise, the same estimator satisfies $|f(q) - \hat{f}(q)| \leq \epsilon N$ with probability at least $1-\delta$.*

The intuition behind the analysis is that $\hat{f}(q)$ has two sources of errors: (a) underestimation or overestimation due to min-wise sampling, and, (b) (one-sided) overestimation due to hash collisions in the outer Count-Min structure. We first provide a bound on the sampling error using the analysis in [18], which is oblivious to hash collisions, and then integrate the error due to hash collisions, of Lemma 1. See Theorem 3.3 and Corollary 3.4 in [19] for the full proof of Theorem 2.

Complexity.

The space complexity of $\mathbb{S}1$ is $C = O(w \times d \times B \times b \times |\mathcal{A}|)$. Computational complexity for query execution is $O(p \times d \times B \times \log(B))$, and for insertions is $O(|\mathcal{A}| \times d \times \log(B))$ – the last logarithm is for maintaining an ordered set of samples, which speeds up insertions.

Configuring the sketch.

The user sets the available memory M, and the values of ϵ and δ. The values of w and d are computed as follows: $w = 1 + \lceil e * ((\epsilon+1)/\epsilon)^{1/d} \rceil = \Theta((1/\epsilon)^{1/d})$ and $d = \lceil \ln(2/\delta) \rceil$. In order to not exceed the memory quota M (in bits), the user chooses the maximum B that satisfies $M \geq w * d * |\mathcal{A}| * (32 + B * (\lceil \log(4B^{2.5}/\delta) \rceil + 3 * 32 + 1))$. [3]

Support for range queries.

Range queries can be enabled by using tree-based structures as dyadic intervals [9] to decrease the query time, similar to how they are used in supporting range queries for the Count-Min sketch. We describe this process in more detail and provide examples in [19]. Maintenance of the internal range sketches increases the space and time complexity of the sketch by a factor of $\log(|\mathcal{D}(a_i)|)$. The approximation error depends on the number of dyadic ranges contained in the canonical cover of the query, which is at most $2\log(|\mathcal{D}(a_i)|)$, as in [9]. Therefore, the total error is at most $2\epsilon\log(|\mathcal{D}(a_i)|)N$.

4. EXPERIMENTAL EVALUATION

The purpose of our experiments was twofold: (a) to compare the space complexity, efficiency, and accuracy of the three proposed sketches to each other and to the state-of-the-art, and, (b) to examine how our best performing sketch, $\mathbb{S}1$, performs when summarizing streams of different characteristics, in different configurations, and for a variety of query types. The detailed experimental setup, a comprehensive evaluation of all methods on multiple datasets, and an in-depth discussion of the results is included in [19], whereas all code and detailed reproducibility instructions can be downloaded from our github.[4] In the following we will highlight the key insights from our experiments, focusing on $\mathbb{S}1$, which was shown to be our best-performing proposed sketch.

[3] To the best of our knowledge, there is no closed-form solution of this formula. We approximate the solution efficiently using the bisection method.

[4] https://github.com/wiegerpunter/omnisketch

Comparison with the state-of-the-art.

We start by comparing $\mathbb{S}1$ with Hydra [16], the only other sketch that supports queries on multiple attributes. For this comparison, we configured both sketches to use the same RAM, and used them to summarize the SNMP dataset [10] which contains a total of 8.2 Million records, with 11 attributes. We compared the two sketches on their ingestion time, query execution performance, and estimation accuracy.

Ingestion time. Since high throughput is the key requirement for stream processing, we first assess the time required by $\mathbb{S}1$ and Hydra to summarize the entire stream, and examine how this is affected by the number of attributes in the stream. We generated streams with varying number of attributes (from 2 to 11) by taking distinct vertical partitions of SNMP (i.e., different subsets of the available attributes). Figure 3 shows the ingestion time of Hydra and $\mathbb{S}1$ for different memory quotas, ranging from 10 to 200 MB. For illustration purposes, the Y axis is split to two sub-ranges of different scales. Our first observation is that both algorithms have comparable performance when storing up to 4 attributes. However, the computational complexity of Hydra grows exponentially with the number of stream attributes. This happens because each record in Hydra leads to $2^{|\mathcal{A}|}-1$ sketch updates, where $\mathcal{A}$ denotes the set of stream's attributes. Therefore, storing all 11 attributes of SNMP in Hydra leads to $O(2^{11})$ updates per record, and takes around 14 thousand seconds for the full dataset (more than 1 msec per record). This result highlights the importance of solutions that avoid this exponential dependency to the number of attributes.

Interestingly, addition of more attributes has a negligible impact on the ingestion time of $\mathbb{S}1$. This stability may appear counter-intuitive at first sight, given that more attributes need to be summarized. The root cause of this result is that the sample size B per cell gets reduced with more attributes in the stream, in order to satisfy the memory constraints. This reduction leads to: (a) reduction of the probability that a sample needs to be updated, and, (b) faster updates of the samples, whenever these are required. Therefore, $\mathbb{S}1$ is suitable for efficiently summarizing complex, multi-attribute streams.

Accuracy. Figure 4 plots the observed error on the two sketches, for queries with $p = 2$, 3, and 4 predicates, and for a varying number of stored attributes. The observed error is the average error, normalized by the length of the stream. We see that even though the number of query predicates does not significantly influence the accuracy of either method, the number of attributes in the dataset does influence the accuracy of Hydra. Beyond 8 attributes in the dataset, Hydra's accuracy is reduced, whereas the accuracy of $\mathbb{S}1$ is not affected, and the average observed error is also very small. This behavior of Hydra is again attributed to the number of inserts that Hydra eventually performs per record: the information summarized by Hydra increases exponentially to the number of attributes, leading to more collisions and to a drastic increase of the observed error.

[5] For illustration purposes, the plot is broken to two different sub-plots, with different scaling at the Y axis.

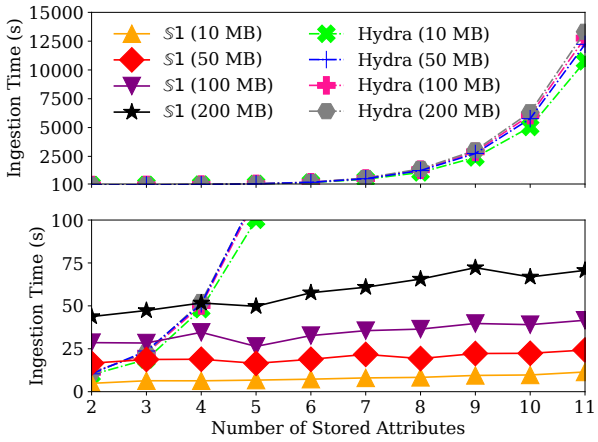

Figure 3: Ingestion Time for Hydra and 𝕊1, for ingesting different vertical partitions (varying number of attributes) of the SNMP stream. [5]

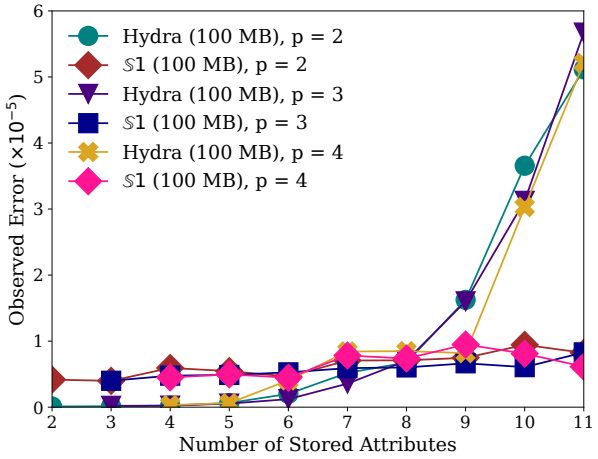

Figure 4: Estimation error of Hydra and 𝕊1, on different vertical partitions of SNMP.

Table 2: Query Execution Time in milliseconds, for queries with different numbers of predicates p.

# query pred.	2	4	6	8
$\mathbb{S}1(10MB)$	0.18	0.13	0.12	0.09
$\mathbb{S}1(50MB)$	0.98	0.79	0.68	0.44
$\mathbb{S}1(100MB)$	2.02	1.61	1.25	1.13
$\mathbb{S}1(200MB)$	6.38	4.33	2.85	2.64
Hydra (10 MB)	0.04	0.04	0.02	0.02
Hydra (50 MB)	0.05	0.04	0.03	0.00
Hydra (100 MB)	0.05	0.04	0.02	0.02
Hydra (200 MB)	0.05	0.04	0.02	0.00

Query execution time. For the final comparison, we used both Hydra and 𝕊1 to summarize all 11 attributes of the SNMP dataset. Table 2 summarizes the query execution time for queries with up to 8 predicates. Both sketches are very efficient, requiring less than 10 msec for executing each query, even in the configuration with a 200 MB RAM quota. Hydra is faster than 𝕊1, since 𝕊1 needs to compute the intersection of $p \times d$ samples of size B, whereas Hydra only needs to compute the minimum of an approximately equal number of counters. For the same reason, the query time for 𝕊1 also grows slightly with the available memory, whereas Hydra performance stays unaffected. Also notice that 𝕊1's efficiency increases with the number of predicates. This improvement is attributed to the algorithm's way of computing the sets intersection, which becomes more efficient as the number of attributes/sets increases (see Section 3.2).

Additional results.

In [19] we presented additional comparisons and sensitivity experiments for 𝕊1, while varying: (a) the stream length – up to 110 Million updates, (b) the dataset characteristic distributions, (c) the query type – range queries, queries with different number of attributes, and, (d) the configuration and memory quota of the sketches. Our results demonstrated that 𝕊1 shows stable performance in all different configurations.

Concluding remarks on experiments.

In conclusion, the exponential dependency on the number of attributes makes sketches like Hydra unsuitable for summarizing complex streams efficiently, as can be seen in Figures 3 & 4. In contrast, 𝕊1, which does not exhibit this dependency, enables efficient summarization and query execution and offers an attractive trade-off between accuracy and efficiency.

5. ONGOING AND FUTURE WORK

In this section, we discuss our vision on future work. We start with a brief outline of our planned extensions for the OmniSketch work, and continue with a broader reflection on the area of general-purpose sketches.

OmniSketch Extensions.

OmniSketch currently supports the typical *insert-only* (i.e., cash register [17]) data stream model, which allows only positive updates to the stream. It can, however, be extended to handle the more general turnstile model, which also supports updates on existing data and deletes. The key challenge lies in the maintenance of the samples contained in each cell of the sketch. A more complex stream-sampling method, like 2LHS [7], can be employed in place of min-wise hashing to support a specific type of deletions. We are also working towards adding support for a variety of query types, such as joins and other L-norms, by appropriately adapting the estimation procedure. Finally, we are examining whether the key idea of OmniSketch – first hashing on the attribute values, followed by consistent hashing/sampling on the record ids – could be generalized to other sketch structures, e.g., the Universal sketch [15], leading to a *sketching framework* for supporting arbitrary predicates on complex streams without the exponential dependency on the number of attributes.

Towards general-purpose sketching solutions.

Our work on OmniSketch is part of our long-term effort to make approximate query processing techniques transparent and accessible to the average data scientist. Despite key advancements on sketches over the last two decades, these

techniques remain not widely adopted, even though they could be used to offer a level of analysis that is currently impossible at a very favorable cost/performance trade-off. We believe that the key barrier for widespread adoption of sketches is the steep learning curve of understanding and using them correctly; this includes, for instance, selecting the best sketch for the problem at hand out of the hundreds of available sketches, correctly configuring the sketch, and interpreting its estimates and error guarantees. This situation is actually reminiscent of the early days of traditional data management systems, where our community lacked a unifying formal model (algebra/calculus), to integrate data processing algorithms and automate optimizations.

Inspired by the historical progress and ease of use of relational databases, we believe that the key to making sketches accessible and boosting their influence on real systems lies in *decoupling user requirements from all sketching-related choices*. This can be achieved by the use of a high-level, declarative (e.g., SQL-like) query language, which can be automatically mapped to an algebra that allows us to reason about sketches and their combinations, and can be mapped down to streaming query execution plans and executable code. We believe that this can give rise to a new wave of off-the-shelf stream processing systems, allowing cost-conscious analytics over fast-paced streams, enabling novel streaming applications, and unlocking exciting optimization opportunities.

6. CONCLUSIONS

We presented OmniSketch, a sketch focused on summarizing the distributions of complex streams (with many attributes) in small space. The sketch combines small (user-defined) memory footprint, and fast updating and querying times (log-linear complexity) and offers theory-backed accuracy guarantees for both point and range queries. A thorough experimental evaluation of the sketch revealed that it can achieve very fast update rates, even when summarizing streams with many attributes. For example, a sketch utilizing 200 MB can summarize an 11-attributes stream with a throughput exceeding 89 thousand updates per second, whereas the 100 MB sketch on the same stream supports twice this throughput. Furthermore, we have shown that OmniSketch outperforms the state-of-the-art (in our experiments, by more than 2 orders of magnitude in throughput) while still providing highly accurate estimates.

Acknowledgements

This work was partially funded by the European Commission under the STELAR (HORIZON-EUROPE - Grant No. 101070122) and DataTools4Heart (HORIZON-HLTH - Grant No. 101057849) projects.

7. REFERENCES

[1] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.

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

[3] Graham Cormode, Minos N. Garofalakis, Peter J. Haas, and Christopher M. Jermaine. Synopses for massive data: Samples, Histograms, Wavelets, Sketches. *Found. Trends Databases*, 4(1-3):1–294, 2012.

[4] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the Count-Min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.

[5] Otmar Ertl. SetSketch: Filling the gap between MinHash and HyperLogLog. *Proc. VLDB Endow.*, 14(11):2244–2257, 2021.

[6] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete Mathematics & Theoretical Computer Science*, Proceedings vol. AH, AofA 07, 2007.

[7] Sumit Ganguly, Minos N. Garofalakis, and Rajeev Rastogi. Tracking set-expression cardinalities over continuous update streams. *VLDB J.*, 13(4):354–369, 2004.

[8] Phillip B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB 2001*, page 541–550.

[9] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin J. Strauss. How to summarize the universe: Dynamic maintenance of quantiles. In *VLDB 2002*, page 454–465.

[10] David Kotz, Tristan Henderson, Ilya Abyzov, and Jihwang Yeo. CRAWDAD Dartmouth/campus (v. 2009-09-09), 2022.

[11] Jakub Lemiesz. On the algebra of data sketches. *Proc. VLDB Endow.*, 14(9):1655–1667, 2021.

[12] Jakub Lemiesz. Efficient framework for operating on data sketches. *Proc. VLDB Endow.*, 16(8):1967–1978, 2023.

[13] Ping Li and Arnd Christian König. b-Bit minwise hashing. In *WWW 2010*, pages 671–680.

[14] Ping Li, Art B. Owen, and Cun-Hui Zhang. One permutation hashing for efficient search and learning. *CoRR*, abs/1208.1259, 2012.

[15] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *SIGCOMM 2016*, page 101–114, 2016.

[16] Antonis Manousis, Zhuo Cheng, Ran Ben Basat, Zaoxing Liu, and Vyas Sekar. Enabling efficient and general subpopulation analytics in multidimensional data streams. *Proc. VLDB Endow.*, 15(11):3249–3262, 2022.

[17] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

[18] Rasmus Pagh, Morten Stöckel, and David P. Woodruff. Is min-wise hashing optimal for summarizing set intersection? In *PODS 2014*, page 109–120.

[19] Wieger R. Punter, Odysseas Papapetrou, and Minos Garofalakis. OmniSketch: Efficient multi-dimensional high-velocity stream analytics with arbitrary predicates. *Proc. VLDB Endow.*, 17(3):319–331, 2023.