OmniSketch: Efficient Multi-Dimensional High-Velocity Stream Analytics with Arbitrary Predicates

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

Odysseas Papapetrou
Eindhoven University of Technology
Eindhoven, The Netherlands
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, or desired 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. In this work, 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 guarantees, a favorable space-accuracy tradeoff, and a worst-case logarithmic complexity for updating and for query execution. We demonstrate experimentally with both 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.

PVLDB Reference Format:
Wieger R. Punter, Odysseas Papapetrou, and Minos Garofalakis. OmniSketch: Efficient Multi-Dimensional High-Velocity Stream Analytics with Arbitrary Predicates. PVLDB, 17(3): 319 - 331, 2023.
doi:10.14778/3632093.3632098

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/wiegerpunter/omnisketch.

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. When it comes to streaming data that cannot be stored or real-time queried in its entirety, the go-to techniques are based on *sketches* [5]: small-memory data structures that summarize the stream and can subsequently be used to execute aggregate queries. Example sketches from the literature support estimation of counts, norms, and join aggregates [2, 6, 23]), estimation of set sizes [9], and identification of frequent items and heavy hitters [4].

Despite being heavily used, most sketches to date focus on summarizing the frequency distribution based on a single attribute, or *a pre-chosen combination* of attributes. Consider, for example, the domain of network monitoring, where the Count-min sketch finds frequent applications for statistics maintenance [6]. A standard IPv4 packet header defines at least 13 fields/attributes, including version, header length, total length, the DSCP code point, source and destination address, protocol, and possibly one or more of 30 additional options. To be able to estimate the number of packets that satisfy a combination of attribute values (defined at query time), we need to construct a sketch that uses as a key the combined values on these attributes (e.g., their concatenation). As a running example, consider the simplified IPv4 headers stream of Figure 1, which contains 4 of the 13 attributes. To summarize the distribution of the number of packets sent by each IP address, we need one sketch constructed using `ipSrc` as the key. One additional sketch on `ipDest` is needed for summarizing the number of packets received by each IP address. If we also want to summarize the number of packets exchanged between any two IP addresses, we need to maintain a sketch that uses the concatenation of the source-destination IP addresses as the key. The number of sketches that need to be maintained to support arbitrary predicates (in this example, any sub-combination of the 13 fields contained in the header) totals to $O(2^{13})$ – the size of the powerset. Generalizing this to arbitrary use cases, estimating the frequency distribution for all sub-combinations of p predicates requires maintenance of $2^p - 1$ sketches. This is clearly infeasible, both because of space requirements and strict efficiency constraints that arise in the context of data streams. [1]

Our Contributions. In this work, we propose, analyze, and evaluate a novel sketching tool, termed OmniSketch, that effectively addresses both space and time efficiency by combining sketching with sampling. OmniSketch, 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, an OmniSketch for summarizing a p-attribute data stream consists of p individual small-memory sub-sketches, each similar

Proceedings of the VLDB Endowment, Vol. 17, No. 3 ISSN 2150-8097.
doi:10.14778/3632093.3632098

[1] The recently proposed Hydra sketch [20] for multi-dimensional streams, addresses space constraints by hashing/summarizing the contents of all 2^p sub-sketches in the same sketch space. Still, as we also discuss later, Hydra's sketch maintenance complexity remains $O(2^p)$, which is simply not viable in real-world streaming use cases, where the number of attributes p can be high (e.g., tens or hundreds).

to a Count-Min sketch. However, unlike Count-Min sketches, the cells in the OmniSketch sub-sketches contain fixed-size summaries of all records that hash into them. At query time, the sub-sketches that are relevant to the query, and the relevant cells from each sub-sketch, are located and queried to estimate the answers. 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. Our sketch is backed by a theoretical analysis for providing formal error guarantees, and an automated initialization algorithm that builds on the theoretical analysis to fully utilize the available sketching 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 complexity-accuracy tradeoff. 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 5.

2 PRELIMINARIES AND RELATED WORK

Preliminaries. OmniSketch inherits the basic structure of Count-Min sketches [6], and builds on the theory of k-minwise hashing [22]. We will now briefly present these two works, to the depth required for understanding our work.

Count-Min sketch. The Count-Min sketch, proposed by Cormode and Muthukrishnan in 2005 [6], became the de facto sketch for the summarization of distributions of data streams. A Count-Min sketch CM is a 2-dimensional array of width w and depth d, accompanied by d pairwise-independent hash functions that map the input to the range $\{1, \dots w\}$. Let $CM[j, h_j(\cdot)]$ denote the counter at row j, column $h_j(\cdot)$ in the 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 $j \in \{1 \dots d\}$. The number of arrivals of any query q in the stream is estimated by finding the corresponding cells $CM[j, h_j(q)]$ per row j and returning the minimum count, i.e., $\hat{f}(q) = \min_{j \in \{1 \dots d\}}(CM[j, h_j(q)])$. 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.

Count-Min sketches also support range queries, by decomposing ranges to canonical covers [6]. Adding support for range queries for an attribute a increases the space and time complexity roughly by a factor of $O(\log(|\mathcal{D}(a)|))$, where $\mathcal{D}(a)$ denotes the attribute's domain.

Due to the attractive cost-accuracy tradeoff of Count-Min sketches, it is also possible to maintain multiple sketches on fast-paced data streams, each summarizing one attribute. For example, the stream of our running example (Figure 1) could be summarized on 4 individual Count-min sketches, thereby enabling frequency estimations with any one of the following four filtering conditions: `WHERE ipSrc=?`, `WHERE ipDest=?`, `WHERE totalLen=?`, `WHERE dscp=?` (with ? denoting the predicate value). However, Count-min sketches do not support multiple predicates in the same query, e.g., `WHERE ipSrc=? AND ipDest=?`. If these predicate combinations are known before observing the stream, then a single sketch can be built for each of these combinations, indexing the concatenation of the attributes. For example, to support the query `WHERE ipSrc=? AND ipDest=?`, we can construct a *composite* key for each record `<ipSrc,ipDest>`, and summarize these in a sketch. Then, queries that have both attributes in the predicates set can be answered by constructing the query's composite key in a similar way, and querying for it. However, if these predicate combinations are not known beforehand, or if we want to enable the user to query with arbitrary attribute combinations, summarization of a stream with n attributes requires the construction of $2^n - 1$ sketches, for covering all possible combinations of predicates. Therefore, this approach is not a viable solution, in terms of both time and space complexity. [2]

K-minwise hashing. In this work, we rely on minwise hashing to estimate the cardinality of the intersection of multiple sets. The key idea behind all minwise 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 K-minwise hashing algorithm that we will be using in this work estimates the sets intersection cardinality as follows [22]: Let $R_1, R_2, \dots, R_p$ 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 $|R_\cap| = |R_1 \cap \dots R_p|$ 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 K-minwise hashing over other sampling methods, since this algorithm has been shown to perform equally well or outperform other sampling methods (including [3, 16, 18]), and has space complexity that nearly matches the theoretical lower bound for the problem [22]. Furthermore, the chosen algorithm also works with streaming data, as the samples can be maintained incrementally.

[2]Note that building a Count-Min sketch on the composite key comprising all n attributes is also not a viable solution. Besides the obvious dimensionality curse problem for large n, such a sketching structure offers no space- or time-efficient way of handling "don't care" attributes (for which no value constraints are specified) when querying with dynamic predicates on arbitrary attribute subsets. Possible approaches would still require a space/time blowup that is exponential in n.

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`

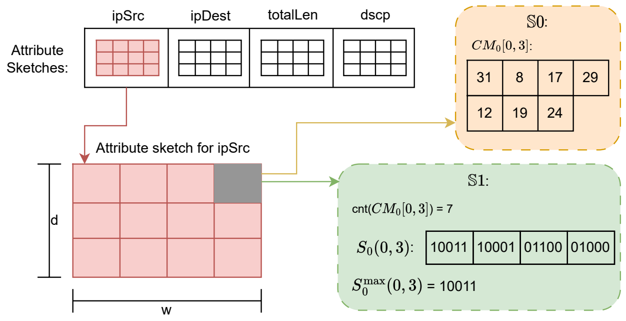

Figure 2: OmniSketch. The yellow-shaded box illustrates the contents of a cell in $\mathbb{S}0$. The green-shaded box corresponds to the contents in $\mathbb{S}1$, for $B = 4$ and signature size set to 5 bits.

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 [21], where a query is posed any time while the stream is ingested and refers to all stream arrivals until that time. [3] 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 $\mathcal{A}$:

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

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 [20]. 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 [19], for keeping detailed statistics. When a record of p attributes is read, it is hashed in the Hydra sketch 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. Distinct sampling was proposed for estimating the number of distinct items in a stream, based on a small-memory sample of the stream [11]. The key idea is to maintain samples at different levels, with a total memory budget B. A record is included in each sampling level l with a probability 2^{-l}. When the total number of samples (across all levels) exceeds B, the lowest level is dropped, thereby releasing approximately half the memory and making space for more samples at higher levels. An (ϵ, δ) estimate of the number of distinct items in the stream can be computed by multiplying the number of items in the smallest surviving sampling level l_{min} with a scale factor $2^{l_{min}}$. Distinct sampling can also be used for estimating the size of set intersection, by exploiting coordinated sampling. In our context, distinct sampling could be used as an alternative of K-minwise hashing to progressively maintain stream samples with a fixed memory budget. In practice, K-minwise hashing was shown to better exploit the available memory in our experiments. Therefore, we do not report experiments with distinct sampling.

The 2-Level hash sketch (2LHS) [10] was proposed for cardinality estimation of arbitrary set expressions, on streams that contain general updates (record insertions and deletions). A 2LHS $\mathcal{X}_A$ comprises two levels of buckets and is implemented as a three-dimensional array of size $k \log(M) \times s \times 2$, where k and s are user-tunable parameters that control the estimation accuracy, and M is the domain size of the input. Conceptually, the 2LHS can be viewed as a generalization of distinct sampling that can be employed to give (ϵ, δ)

[3]We briefly discuss extending OmniSketch to handle general updates (i.e., record deletes as well as inserts) later in the paper.

cardinality estimates of set unions, intersections, and differences over general update (i.e., turnstile [21]) data streams. While 2LHS could be used as an alternative for K-minwise hashing, there is added space/time complexity in the 2LHS structure, which is actually necessary for handling record deletions in the stream. Thus, we focus our OmniSketch design on K-minwise hashing which offers a much simpler and more space-efficient solution for insert-only streams, and discuss possible extensions for general updates later in the paper.

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 [7, 8, 14, 15, 17]. At their core, these methods are similar to K-minwise hashing, with similar complexities and theoretical guarantees. In principle, any of these could also be used with OmniSketch to construct the nested sketch. It is important to note, however, that none of these methods (including K-minwise 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: ESTIMATING FREQUENCIES 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 OminiSketch ($\mathbb{S}1$) which merges $\mathbb{S}0_{\cap}$ with a sampling technique to guarantee sublinear space complexity. Unlike previous works, all three sketches allow very efficient updates (with logarithmic complexity), and can therefore easily handle fast-paced streams.

All three sketches are based on a common architecture: each 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 $CM_1, CM_2, \ldots CM_{|\mathcal{A}|}$, 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, $p \leq |\mathcal{A}|$. We summarize frequently used notation in Table 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.
$R(CM_i[j,k])$	Set of records hashed to cell $CM_i[j,k]$.
$S_i(j,k)$	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.
$R_{\cap}$	Set of records hashed to all cells in $C(q)$.
$n_{\max}$	Max. number of records hashed to any cell $CM_i[j,k] \in C(q)$.

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 $h_i^j() : \mathcal{D}(a_i) \rightarrow \{1, \ldots, w\}$. 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 $d = \lceil \ln(1/\delta) \rceil$.

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.

The $\mathbb{S}0_{\min}$ Estimator. Let $f(q)$ denote the number of records satisfying all predicates in q. Following the conventional Count-Min estimation procedure [6], we can compute an estimate $\hat{f}(q)$ as follows:

for each row $j = \{1, \ldots, d\}$, we compute the size of the intersection of all records stored in cells $CM_1[j, h_1^j(v_{q_1})], CM_2[j, h_2^j(v_{q_2})], \ldots, CM_p[j, h_p^j(v_{q_p})]$. We repeat this for the d rows, and return the minimum value as an estimate. Formally:

$$\hat{f}(q) = \min_{j=\{1 \ldots d\}} \left| \bigcap_{q_i \in q} CM_i[j, h^j(v_{q_i})] \right| \tag{1}$$

Intuitively, the intersection will contain the rids of all records r that satisfy the query, since these will always hash at the same cell as the query predicate, for all predicates, However, the intersection may also contain some false positives, i.e., records that were hashed in $CM_i[j, h_i^j(v_{q_i})]$ due to one or more random collisions. By taking the minimum intersection size across all d rows, we try to minimize the effect of such false positives.

Error bounds. The following lemma provides probabilistic guarantees for the estimator of Eqn. 1.

Lemma 3.1. *Let $\hat{f}(q)$ be the estimate provided by Equation 1 on sketches constructed with $d = \lceil \ln(\frac{1}{\delta}) \rceil$, $w = 1 + \lceil \frac{e}{\epsilon} \rceil$. For query q with p predicates:*

$$Pr\left(|\hat{f}(q) - f(q)| \leq \epsilon N\right) \geq 1 - \delta \tag{2}$$

Proof. Consider row $j \in \{1 \ldots d\}$. Let us define an indicator variable $I_{y,j}$, which takes the value of 1 for all records y satisfying the following condition:

$$\forall_{q_i \in q}[h_i^j(v_{q_i}) = h_i^j(y_i)] \wedge \exists_{q_i \in q}[v_{q_i} \neq y_i]$$

Informally, the described condition is true for all false positive records, i.e., records that are contained in the intersection but do not fully satisfy the query. Also, variable $X^j = \sum_{y=1}^{N-f(q)} I_{y,j}$ is a counter of these false positives for row j.

Recall that the estimator for row j is as follows: $\hat{f}^j(q) = |R_\cap^j|$, where $R_\cap^j$ denotes the intersection of the cells at position $h_i^j(v_{q_i})$, for $i = \{1, \ldots, p\}$. We need to prove that: (a) all records that satisfy all predicates will be contained in $R_\cap^j$, i.e., there will be no false negatives, and (b) the total number of false positives X is upper-bounded w.h.p..

For (a), notice that for any record y satisfying $y_i = v_{q_i}$ for $i = \{1 \ldots p\}$, we will have $y_i = v_{q_i} \Rightarrow h_i^j(y_i) = h_i^j(v_{q_i})$. Therefore, all records satisfying all predicates will be included in $R_\cap^j$.

For (b), a record will be a false positive and increase the value of X^j by 1 if for all attributes y_i for which $y_i \neq v_{q_i}$, we have $h_i^j(y_i) = h_i^j(v_{q_i})$. Let X_k^j denote the number of records contained in $R_\cap^j$ that satisfy *exactly* $p - k$ predicates, i.e., they do not satisfy k of the p predicates, but they are retrieved by the algorithm due to hash collisions. Then, $X^j = \sum_{k=1}^{p} X_k^j$.

By construction of the hash functions, a record y with $y_i \neq v_{q_i}$ will have $h_i^j(y_i) = h_i^j(v_{q_i})$ with probability $1/w$. Since the hash functions are pairwise-independent, the probability that a record differing from q in k attributes hashes in the same cells is $1/w^k$. Therefore, $E[X_k^j] \leq (N - f(q))/w^k$, and $E[X^j] = \sum_{k=1}^{p} E[X_k^j] \leq (N - f(q)) * (1/w + 1/w^2 + \ldots + 1/w^p) = (N - f(q)) * (1 - (1/w)^p)/(w - 1)$. By setting $w = 1 + \lceil e/\epsilon \rceil$ we get $E[X] = (N - f(q))/e * (1 - \epsilon^p)/\epsilon \leq \epsilon(N - f(q))/e \leq \epsilon N/e$.

Notice that we have d rows, each with different hash functions. By Markov's inequality, we have

$$\begin{aligned} Pr[X \geq \epsilon(N - f(q))] &= Pr[\forall_{j \in [1 \ldots d]} . \hat{f}^j(q) > f(q) + \epsilon N] \\ &= Pr[\forall_{j \in [1 \ldots d]} . X^j > eE[X]] \\ &< e^{-d} \end{aligned}$$

where $f(q)$ denotes the true answer and X^j denotes the false positives at row r, i.e., $X_j = \hat{f}^j(q) - f(q)$. The final bound follows by setting $d = \lceil \ln(1/\delta) \rceil$. □

$\mathbb{S}0_\cap$: An Improved Estimator. The $\mathbb{S}0_{\min}$ estimator is based on the conventional estimation logic in the standard Count-Min sketch [6]; that is, it produces an estimate per row and takes the minimum across all the d row estimates. 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 only end up in a few of the rows. Thus, without modifying the sketch construction or space/time complexity, we can get a tighter estimator by removing the min operation across rows and simply intersecting 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:

$$\hat{f}(q) = \left| \bigcap_{i=\{1 \ldots p\}, j=\{1 \ldots d\}} CM_i[j, h_i^j(v_{q_i})] \right| \tag{3}$$

Clearly, this new $\mathbb{S}0_\cap$ estimator can also only overestimate 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 in Eqn. 1. We now demonstrate the stronger error guarantees of $\mathbb{S}0_\cap$, proving that it allows us to bound the error by $\epsilon^d(N - f(q))$ instead of $\epsilon(N - f(q))$, without changing the space complexity of the sketch.

Lemma 3.2. *Let $\hat{f}(q)$ be the estimate provided by Equation 3 on sketches constructed with $d = \lceil \ln(\frac{1}{\delta}) \rceil$, $w = 1 + \lceil \frac{e}{\epsilon} \rceil$. For query q with p predicates:*

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

Proof. The proof is similar to the proof for Lemma 3.1.

We define an indicator variable I_y, which takes the value of 1 for all records y satisfying the following condition:

$$\forall_{q_i \in q}[\forall_{1 \leq j \leq d}[h_i^j(v_{q_i}) = h_i^j(y_i)] \wedge \exists_{q_i \in q}[v_{q_i} \neq y_i]]$$

Informally, I_y becomes 1 for all false positive records, i.e., the records that hash to the same cells as the query predicate value, at *all d* rows and at *all p* predicates. Also, $X = \sum_{y=1}^{N-f(q)} I_y$ is a counter of the total number of false positive records.

We will prove that (a) all records that satisfy all predicates will be contained in $\bigcap_{j=1}^{d} R_{\cap}^{j}$, i.e., there will be no false negatives, and, (b) the number of false positives X is upper-bounded w.h.p..

For (a), notice that for any record y satisfying $y_i = v_{q_i}$ for $i = \{1 \dots p\}$, we will have $y_i = v_{q_i} \Rightarrow \forall_{1 \le j \le d}[h_i^j(y_i) = h_i^j(v_{q_i})]$. Therefore, all records satisfying all predicates will be included in the intersection of all $R_{\cap}^{j}$, denoted as $R_{\cap}$.

For (b), a record y will increase the value of X if for all attributes y_i for which $y_i \neq v_{q_i}$, and for all rows $j \in \{1 \dots d\}$, it collides with $h_i^j(v_{q_i})$, i.e., $h_i^j(y_i) = h_i^j(v_{q_i})$. The probability of this collision at a single row j is $Pr[h_i^j(y_i) = h_i^j(v_{q_i})] = 1/w$. Furthermore, by independence of the hash functions across sketch rows, the probability that a record collides with the query at all d rows j will be $1/w^d$.

Let X_k denote the number of records from $R_{\cap}$ that satisfy *exactly* $p - k$ predicates. Then, $E[X_k] \le (N - f(q))/w^{kd}$, and $E[X] = \sum_{k=1}^{p} E[X_k] \le (N - f(q)) \sum_{k=1}^{p} 1/w^{kd} = (N - f(q)) \frac{1-w^{-d(p+1)}}{w^d - 1} < (N - f(q))/(w^d - 1)$. By setting $w = 1 + \lceil e/\epsilon \rceil$, we get $E[X] \le (N - f(q))/((1 + e/\epsilon)^d - 1) \le \epsilon^d (N - f(q))/e^d$.

The following bound follows directly from Markov inequality:

$$Pr[X \ge \epsilon^d (N - f(q))] = Pr[X \ge e^d E[X]] \le e^{-d}$$

Then, by setting $d = \lceil \ln(1/\delta) \rceil$ we get our final bound, $Pr[X \ge \epsilon^d (N - f(q))] \le \delta \Rightarrow Pr[X \ge \epsilon^d N] \le \delta$ □

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 K-minwise hashing algorithm [22]. We now describe our OmniSketch 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 explain how these values are determined later in this section.

Similar to $\mathbb{S}0$, $\mathbb{S}1$ is composed of a collection of $|\mathcal{A}|$ attributes sketches. Furthermore, $\mathbb{S}1$ incorporates a hash function $g : \mathcal{D}(r_0) \to \{0,1\}^b$, which maps each record id 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 K-minwise hashing (Section 2). The $|\mathcal{A}|$ attributes sketches $CM_1, CM_2, \dots CM_{|\mathcal{A}|}$ 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 minwise 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 includes 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 minwise 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 $S_i^{\max}(j,k) = \infty$.

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 $C(r) = \{CM_i[j, h_i^j(r_i)] : i = \{1, \dots, |\mathcal{A}|\}, j = \{1, \dots, d\}\}$. 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 $S_i^{\max}(j,k)$. [4] 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 $i \in \{1 \dots |\mathcal{A}|\}$ do
3:     for $j \in \{1 \dots d\}$ 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 $S_i^{\max}(j,k)$
```

OmniSketch 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., $C(q) = \{CM_i[j, h^j(v_{q_i})] : i = \{1 \dots p\}, j = \{1 \dots d\}\}$, and $n_{\max} = \max_{CM_i[j,k] \in C(q)} cnt(CM_i[j,k])$ be the maximum count of these cells. Following the reasoning of our improved $\mathbb{S}0_{\cap}$ estimator, let

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

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

[4]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.

$$\hat{f}(q) = \frac{n_{\max}}{B} \times |S_\cap| \quad (4)$$

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 K-minwise hashing [22], our estimator comes with a sanity bound of $\frac{3n_{\max}\log(4pd\sqrt{B}/\delta)}{B\epsilon^2}$ to cover the case of insufficient samples in the intersection.

Efficient computation of the intersection. Computing $|S_\cap|$ requires a multi-way join over all samples that participate in the query. For large values of B, a naive hash-based computation of this intersection can take hundreds of milliseconds. To speed up query execution, we exploit the fact that the samples at each cell are already stored in a red-black tree, which allows for sorted iteration and search with complexity logarithmic in B. Our code is an extension of the sort-merge join for multi-way joins. Starting from the first cell, we get the first sample (the one with the minimum hash value), and execute a lookup on all other cells to check whether this hash value is contained in their samples. As soon as we find a cell that does not contain this sample, we get the smallest sample from that cell with a hash value greater than the one that failed, and resume our search from this value (no other value in between the failed and this value could be part of the join). An interesting observation, which also becomes obvious in our experimental results (see Section 4), is that this algorithm typically becomes more efficient when answering queries with more predicates, because it allows for larger steps. For example, if one of the cells contains very few samples, this cell will lead to skipping many candidate records at all other cells. This can bring the complexity of query execution from $O(p * d * B * \log(B))$ down to effectively $O(B\log(B))$.

Derivation of the error bounds. $\hat{f}(q)$ has two sources of errors: (a) underestimation or overestimation due to K-minwise hash 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, which is oblivious to hash collisions, and then integrate the error due to hash collisions. Recall that (ϵ_1, δ_1) and (ϵ_2, δ_2) correspond to the (ϵ, δ) configurations of the K-minwise sampling algorithm and the attribute sketch respectively.

We use $R(CM_i[j,k])$ to denote all records that are hashed into $CM_i[j,k]$, even if these do not end up in the sample, and $R_\cap = \bigcap_{CM_i[j,k]\in C(q)} R(CM_i[j,k])$ to denote the intersection of records that are hashed in all cells in $C(q)$. Finally, $S_i(j,k)$ denotes the samples collected at cell $CM_i[j,k]$ and $S_\cap$ the intersection of samples of all cells in $C(q)$.

The following theorem and corollary provide the error guarantees for our OmniSketch estimator. (Note that in any realistic setting, $n_{\max} << N$.)

Theorem 3.3. *Consider an OmniSketch with* $\epsilon_1 = \epsilon$, $\epsilon_2 = (\epsilon/(1+\epsilon))^{1/d}$, *and* $\delta_1 = \delta_2 = \delta/2$. *If* $|S_\cap| < \frac{3\log(4pd\sqrt{B}/\delta)}{\epsilon^2}$, *setting* $\hat{f}(q) = \frac{2n_{\max}\log(4pd\sqrt{B}/\delta)}{B\epsilon^2}$ *satisfies* $|f(q) - \hat{f}(q)| < \frac{2n_{\max}\log(4pd\sqrt{B}/\delta)}{B\epsilon^2}$ *with probability at least* $1 - \delta/2$. *Otherwise, setting* $\hat{f}(q) = \frac{n_{\max}}{B} \times |S_\cap|$ *satisfies* $|f(q) - \hat{f}(q)| \le \epsilon N$ *with probability at least* $1 - \delta$.

Proof. We distinguish two cases, based on the value of $|S_\cap|$:

Case 1 We first examine the case that $|S_\cap| \le \frac{3\log(2pd\sqrt{B}/\delta_1)}{\epsilon_1^2}$. From Theorem 4 of [22], and by setting the theorem's δ to $\delta_1/\sqrt{B}$, we derive that $0 \le |R_\cap| \le \frac{4n_{\max}\log(2pd\sqrt{B}/\delta_1)}{B\epsilon_1^2}$. We also know that $f(q) \le |R_\cap|$, since the latter may contain false positives – records that do not fully satisfy all query predicates, but still end up in all cells of $C(q)$ due to hash collisions. Therefore, $f(q) \le \frac{4n_{\max}\log(2pd\sqrt{B}/\delta_1)}{B\epsilon_1^2}$. By returning $\hat{f}(q) = \frac{2n_{\max}\log(2pd\sqrt{B}/\delta_1)}{B\epsilon_1^2}$ we guarantee that $|\hat{f}(q) - f(q)| \le \hat{f}(q)$ with probability $1 - \delta_1$. Furthermore, setting $\delta_1 = \delta/2$ and $\epsilon_1 = \epsilon$ we get the final estimator for this case: $\hat{f}(q) = \frac{2n_{\max}\log(4pd\sqrt{B}/\delta)}{B\epsilon^2}$, and $|\hat{f}(q) - f(q)| \le \frac{2n_{\max}\log(4pd\sqrt{B}/\delta)}{B\epsilon^2}$ with probability $\ge 1 - \delta_1 = 1 - \delta/2$.

Case 2 Now consider the case that $|S_\cap| > \frac{3\log(2pd\sqrt{B}/\delta_1)}{\epsilon_1^2}$. From Theorem 4 of [22]:

$$\frac{n_{max}}{B}|S_\cap| = (1 \pm \epsilon_1)|R_\cap| \quad (5)$$

with probability at least $1 - \delta_1$. Notice however that due to hash collisions, not all records contained in $R_\cap$ (and hence, also in $S_\cap$) will fully satisfy the query. The number of false positives in $R_\cap$ can be estimated using Lemma 3.2: $|FP| = ||R_\cap| - f(q)| \le \epsilon_2^d(N - f(q))$, with probability $1 - \delta_2$.

From Eqn. 5 and the triangle inequality we have

$$\begin{aligned}\left|\frac{n_{max}}{B}|S_\cap| - f(q)\right| \le & \left|\frac{n_{max}}{B}|S_\cap| - |R_\cap|\right| + (|R_\cap| - f(q)) \\ \le & \epsilon_1|R_\cap| + \epsilon_2^d(N - f(q)) \\ \le & \epsilon_1\Big(f(q) + \epsilon_2^d(N - f(q))\Big) + \epsilon_2^d(N - f(q))\end{aligned}$$

with probability at least $1 - \delta_1 - \delta_2$.

Let us set $c = f(q)/N$ for convenience. Then:

$$\begin{aligned}|\frac{n_{max}}{B}|S_\cap| - f(q)| \le & \epsilon_1\Big(cN + \epsilon_2^d(N - cN)\Big) + \epsilon_2^d(N - cN) \\ \le & N[\epsilon_1 c + \epsilon_1\epsilon_2^d(1-c) + \epsilon_2^d(1-c)] \\ = & N\left[\epsilon_2^d(\epsilon_1 + 1) - c(\epsilon_1\epsilon_2^d + \epsilon_2^d - \epsilon_1)\right]\end{aligned}$$

By setting $\epsilon_1 = \epsilon < 1/4$, $\epsilon_2 = (\epsilon/(1+\epsilon))^{1/d}$, $\delta_1 = \delta_2 = \delta/2$, we get $|\frac{n_{max}}{B}|S_\cap| - f(q)| \le \epsilon N$, with probability $1 - \delta_1 - \delta_2$. □

The following corollary simplifies the estimation procedure, by always using the estimator proposed for case 2.

Corollary 3.4. *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* $|f(q) - \hat{f}(q)| \le \frac{4n_{\max}\log(4pd\sqrt{B}/\delta)}{B\epsilon^2}$ *with probability at least* $1 - \delta/2$. *Otherwise, the same estimator satisfies* $|f(q) - \hat{f}(q)| \le \epsilon N$ *with probability at least* $1 - \delta$.

The corollary follows directly from the theorem, by noticing that when $|S_\cap| < \frac{3n_{\max}\log(4pd\sqrt{B}/\delta)}{B\epsilon^2}$, then $f(q) \in \{0, \frac{4n_{\max}\log(4pd\sqrt{B}/\delta)}{B\epsilon^2}\}$

with high probability, and $\hat{f}(q) \in \{0, \frac{4n_{\max}\log(4pd\sqrt{B}/\delta)}{B\epsilon^2}\}$. Therefore, with high probability, the estimator is at most $\frac{4n_{\max}\log(4pd\sqrt{B}/\delta)}{B\epsilon^2}$ away of $f(q)$.

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 $M \geq w * d * |\mathcal{A}| * (32 + B * (\lceil\log(4B^{2.5}/\delta)\rceil + 3 * 32 + 1))$. [5]

3.3 Extensions

Support for range queries. All proposed sketches support range queries on numeric attributes, as well as combinations of range and point predicates, e.g., counting the number of records with $integer(112.1.4.1) \leq ipSrc \leq integer(112.1.255.255)$, $integer(202.21.1.1) \leq ipDest \leq integer(202.22.255.255)$, and $0 \leq dscp \leq 16$. This is achieved by indexing and querying dyadic ranges, similar to the technique used in Count-min sketches [6]. In particular, the ranges-enabled OmniSketch contains $\log(|\mathcal{D}(a_i)|)$ internal attributes sketches for each attribute a_i, denoted as $CM_{(i,0)}, CM_{(i,1)}, \ldots, CM_{(i,\log(|\mathcal{D}(a_i)|)-1)}$. Each internal sketch $CM_{(i,k)}$ keeps the frequency statistics for dyadic ranges of length 2^k. Therefore, $CM_{(i,0)}$ stores statistics for points, $CM_{(i,1)}$ stores statistics for ranges of size 2, and so on.

A record r is indexed into the sketch as follows. For each numerical attribute r_i, and for $k = 0$ to $\log(|\mathcal{D}(a_i)|) - 1$, we find all ranges of the form $[x * 2^k + 1, (x + 1)2^k]$ that contain r_i, where $x \in \mathbb{Z}^*$. For each range of size 2^k, we add x to $CM_{(i,k)}$. Query execution follows a similar process. At query time, any range predicate is broken to its canonical cover – all sub-ranges that follow the above form. Then, for each row of the sketch, we query all sub-ranges and merge the retrieved samples, effectively constructing a single sample that covers the complete query range. The remaining querying process remains identical to point queries.

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 [12]. Therefore, the total error is at most $2\epsilon\log(|\mathcal{D}(a_i)|)N$.

Support for general updates. As described, OmniSketch supports the typical insert-only (i.e., cash register [21]) data stream model. OmniSketch can be extended to handle the more general turnstile model, i.e., with updates on existing data and deletes. Incrementally maintaining the in-bucket samples then becomes the major challenge. If we have fully specified deletes (i.e., all attribute values including the rid are known at the time of deletion), then a more complex stream-sampling method like 2LHS (Section 2) can be employed in place of K-minwise hashing to support deletions. The case where the deletes are not fully-specified (e.g., only some attribute values are known) is more challenging to address in constant/logarithmic time, and is part of our ongoing work.

4 EXPERIMENTAL EVALUATION

The purpose of our experiments was: (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, and in different configurations.

Datasets. For our experiments, we have used two real-world datasets (SNMP [13] and CAIDA [1]), and several synthetic datasets that enabled us to thoroughly evaluate particular properties of our algorithms. SNMP contains 8.2 Million records with 11 attributes, whereas CAIDA contains 109 Million records, each with 6 attributes. The SNMP dataset contains records collected from the wireless network of Dartmouth college during the fall of 2003, whereas CAIDA is a network flow monitoring dataset, collected by an internet service provider in the USA. We also generated synthetic datasets, with different properties/distributions, to check how the properties of the input stream affect our algorithms. Unless otherwise mentioned, the reported results correspond to the SNMP dataset. The code for dataset loading/generation/pre-processing, as well as the list of all files and attributes used in our experiments, is included in our github repository. [6]

Queries. We methodically constructed the queries in order to comprehensively cover a broad spectrum of query characteristics (different number of results, different number of predicates, different predicate values). At a pre-processing step, we went over a small sample of the dataset (roughly 0.05% of all records). For each sampled record, and for each possible number of predicates $p \in [2, |\mathcal{A}|]$, we constructed 10 queries by randomly choosing p attributes from the set of attributes $\mathcal{A}$, and their values in the sampled record as predicates. All queries were maintained in a set, effectively filtering out duplicates. This led to a total of 39,319 queries for the SNMP dataset on 11 distinct attributes, and 3,304 queries for CAIDA on 6 distinct attributes. More details can be found in the github repository.[6].

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. All sketches were implemented and executed in Java (OpenJDK version 19.0.2). For the Hydra baseline, we used the original code of the authors [20]. Since Hydra was originally constructed as a Spark plugin, to avoid unnecessary extra delays imposed by Spark, we extracted and used only the code that was necessary for centralized execution. Also, to ensure a fair comparison, we excluded from Hydra's implementation any code

[5]To the best of our knowledge, there is no closed-form solution of this formula. However, the solution can be efficiently approximated with a numerical computation toolkit, e.g., using the bisection method.

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

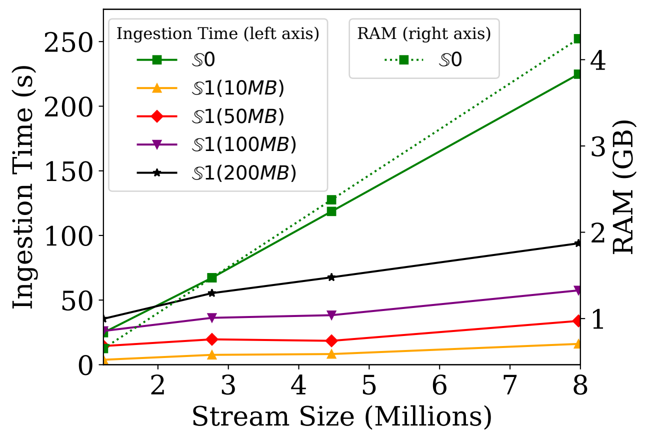

Figure 3: Ingestion time and RAM for summarizing different stream lengths

and data structures associated with statistics beyond cardinality estimation, such as the L2 norm and entropy of sub-populations. Our changes improved Hydra's performance and reduced its space complexity. All algorithms were given a few seconds warm-up time (300 thousand updates), before we started to measure ingestion time. The code of our methods, as well as detailed instructions on processing the datasets and reproducing our results are made publicly available. [6]

4.1 Comparison of $\mathbb{S}0$ and $\mathbb{S}1$

Our first series of experiments was designed for comparing the two $\mathbb{S}0$ estimators ($\mathbb{S}0_{\min}$ and $\mathbb{S}0_{\cap}$) with $\mathbb{S}1$, in terms of (a) stream ingestion time, (b) memory requirements, (c) accuracy of the estimates, and, (d) query execution time. Accuracy was quantified by computing the mean absolute error, normalized by the stream size, i.e., $\sum_{q \in \mathbb{Q}} |\hat{f}(q) - f(q)|/(N * |\mathbb{Q}|)$, where N denotes the stream size and $\mathbb{Q}$ denotes the set of executed queries. For this set of experiments, all sketches were configured with ($\epsilon = 0.1, \delta = 0.1$). We tested four different configurations of $\mathbb{S}1$, with 10, 50, 100, and 200 MB of RAM.

Figure 3 shows the required time and memory for summarizing subsets of the SNMP stream (all 11 attributes), with the compared sketches. Recall that $\mathbb{S}0_{\min}$ and $\mathbb{S}0_{\cap}$ use the same sketch, and are therefore presented together as $\mathbb{S}0$ in this figure. $\mathbb{S}1$ memory requirements are not included in the figure as separate series, since these are fixed for each configuration. As expected, we see that the required time for building all sketches (left axis) grows linearly with the stream size. Sketch $\mathbb{S}1$ is however significantly more efficient, requiring 4 to 5 times less time compared to $\mathbb{S}0$. Also, the throughput of $\mathbb{S}1$ decreases when the sketch is allowed to use more memory. This is again expected, since a higher memory quota for $\mathbb{S}1$ translates to a higher value of B (more samples per cell), which leads to slower maintenance of the red-black tree. Still, even for $\mathbb{S}1$ with 200MB RAM and 11 attributes, throughput exceeds 89 thousand updates per seconds. Also notice that $\mathbb{S}1$ ingestion time forms a subtle, yet visible, elbow (e.g., for $\mathbb{S}1$ with 200MB, this happens at around 2.7 Million updates). This elbow signifies that the K-minwise samples at most cells in the sketch reached to an almost *stable* state, and the probability that any new update needs to be added to the samples is small. At this point, most updates take only $O(|\mathcal{A}| \times d)$ time, as opposed to $O(\log(B) \times |\mathcal{A}| \times d)$. Finally, we observe that the memory requirements of $\mathbb{S}0$ (Fig. 3, right axis) grow linearly with the stream size, since all records are kept in the sketch.

Table 2 summarizes the average error and query execution time for all considered configurations. We observe that both $\mathbb{S}0$ variants take significantly more time for query execution compared to the $\mathbb{S}1$ variants. The difference in performance is somewhere between 1 and 2 orders of magnitude, depending on the memory quota of $\mathbb{S}1$. The reason for this stark difference is because the $\mathbb{S}0$ variants need to iterate over very large sets for computing the intersection, whereas $\mathbb{S}1$ iterates over sample sizes of size B, with B in the order of a few tens of thousands. We also see that $\mathbb{S}0_{\cap}$ estimator is notably faster than $\mathbb{S}0_{\min}$, even though both operate on the same sketch structure. This is attributed to the way the query execution is implemented in the two estimators. In $\mathbb{S}0_{\cap}$, similar to $\mathbb{S}1$, we start from the first cell, and keep reducing the candidate records until we intersect all $p \times d$ cells (see the relevant discussion in the implementation of $\mathbb{S}1$, Section 3.2). This quickly reduces the candidate records. On the other hand, the implementation of the $\mathbb{S}0_{\min}$ estimator performs the same process *per row*, and takes the minimum value of all rows. Therefore, $\mathbb{S}0_{\min}$ repeats each check multiple times, for each record that satisfies the query.

In terms of approximation error, $\mathbb{S}0_{\cap}$ outperforms all others, with a negligible average error. This is to be expected, since $\mathbb{S}0_{\cap}$ does not suffer from underestimations and false negatives, and its probability of including false positives in the estimate is very small: $\epsilon^{p'd}$, where p' denotes the number of query attributes not satisfied by the record. We also see that $\mathbb{S}1$, which relies on the intersection, has around an order of magnitude smaller error than $\mathbb{S}0_{\min}$, even though it requires one to two order of magnitudes less memory for representing the same dataset.

Summary. The comparison between $\mathbb{S}0_{\min}$, $\mathbb{S}0_{\cap}$ and $\mathbb{S}1$ revealed that $\mathbb{S}0_{\cap}$ substantially outperforms the other two variants in terms of accuracy, but with linear space requirements. Therefore, $\mathbb{S}0_{\cap}$ is only useful as an alternative to full indexing, where there is sufficient memory for storing the whole stream. The size of $\mathbb{S}0_{\min}$ also grows linearly with the data, and the estimator also performs worse compared to $\mathbb{S}0_{\cap}$. On the other hand, $\mathbb{S}1$ offers a favorable trade-off between accuracy and space complexity/efficiency, and allows the user to fine-tune the memory quota in order to fully utilize the available RAM. Therefore, for the remaining experiments and comparison with the state-of-the-art we will only consider $\mathbb{S}1$.

4.2 Comparison with the state-of-the-art

Our second series of experiments focused on comparing $\mathbb{S}1$, our best-performing sketch, with Hydra. The two sketches were allowed the same memory, and were compared on their: (a) ingestion time, (b) query execution performance, and, (c) accuracy.

Ingestion time. High throughput is the key requirement for stream processing and summarization. Hence, our initial experiment aimed to assess the time required by $\mathbb{S}1$ and Hydra to summarize the entire

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

Table 2: Time required for query execution, and estimation error

Algorithm	Query Time (ms)	Observed Error
$\mathbb{S}0_{\cap}$	20.886	4×10^{-6}
$\mathbb{S}0_{\min}$	29.867	0.0038
$\mathbb{S}1(10MB)$	0.227	0.0004
$\mathbb{S}1(50MB)$	1.212	0.0003
$\mathbb{S}1(100MB)$	2.452	0.0003
$\mathbb{S}1(200MB)$	6.620	0.0003

Table 3: 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

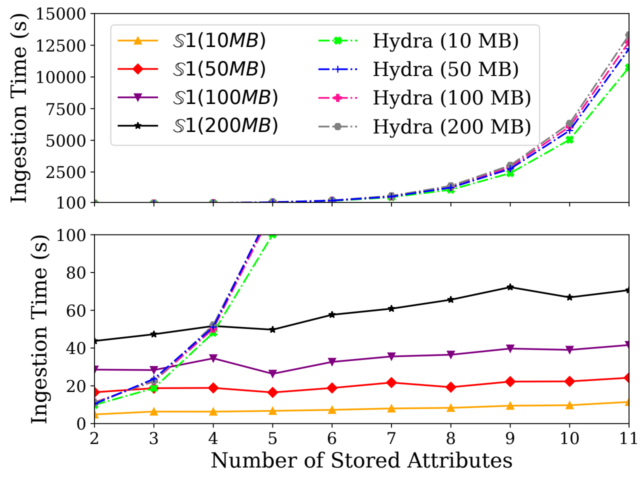

Figure 4: Ingestion Time for Hydra and $\mathbb{S}1$, for ingesting different vertical partitions (varying number of attributes) of the SNMP stream. [7]

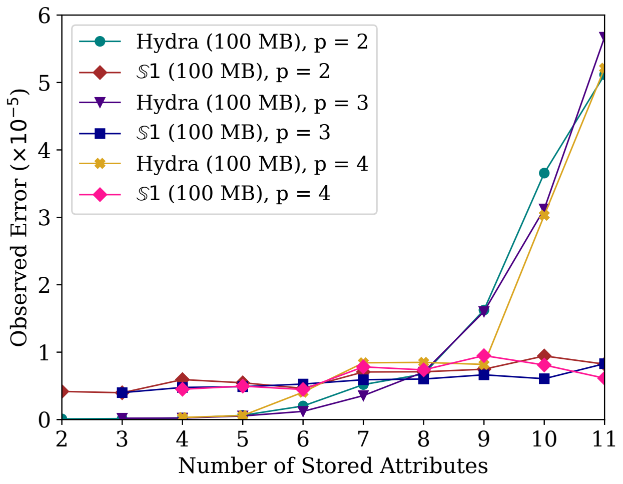

Figure 5: Estimation error of Hydra and $\mathbb{S}1$, on different vertical partitions of SNMP

stream, and to examine how this time was 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 4 shows the ingestion time of Hydra and $\mathbb{S}1$, for different memory quotas. 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 $O(2^{11})$ sub-populations/updates per record, and takes around 14 thousand seconds for the full dataset (more than 1 msec per record). It is important to note here that storing all sub-populations is required in Hydra, not only for supporting queries with exactly 11 predicates, but also for supporting queries that contain subsets of these predicates whenever the exact combinations of the query predicates is not known before observing the stream.

Interestingly, addition of more attributes has a negligible impact on the ingestion time of $\mathbb{S}1$. In fact, there exist cases where adding an attribute does not increase, or even slightly reduces ingestion time (e.g., when adding the fifth attribute for $\mathbb{S}1$, 200 MB). At first sight, this result may appear counter-intuitive, since adding more attributes means maintaining more attribute sketches, and storing more samples in these sketches. This behavior is attributed to the fact that the most time-consuming step of adding a record at $\mathbb{S}1$ is the one of maintaining the K-minwise samples, which becomes faster when the sample size B is reduced. By adding more attributes in the sketch without increasing its memory quota, we effectively reduce the sample size B for all of the sketch's attributes. This leads to: (a) reduction of the probability that the red-black tree needs to be updated, and, (b) faster updates of the red-black tree, whenever these are required.

Accuracy. Figure 5 plots the observed error on the two sketches, for queries with p = 2, 3, and 4 predicates. We see that even though the number of query predicates does not significantly influence the accuracy of either method, the number of stored attributes does influence the accuracy of Hydra. Beyond 8 attributes, Hydra's accuracy is reduced, whereas the accuracy of 𝕊1 is not affected. 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.

Query execution time. For the final comparison, we used both Hydra and 𝕊1 to summarize all 11 attributes of the whole SNMP stream (8.3 Million updates). Table 3 summarizes the query execution time for queries with up to 8 predicates. Both sketches are very efficient, requiring less than 10 msec for executing a query, even in the configuration with a 200MB RAM quota. We also note that 𝕊1 is slower than Hydra. This is expected, 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, aligning with our earlier observation (Section 4.1). 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).

Summary. Both Hydra and 𝕊1 require less than 10 msec to execute queries with up to 8 predicates. However, 𝕊1 substantially outperforms Hydra in efficiency when summarizing streams with many attributes, with a difference in throughput that may exceed two orders of magnitude. Importantly, Hydra's throughput rate decreases exponentially with the number of stream attributes. The same trend of exponential decrease with the number of stream attributes is observed on Hydra's estimation accuracy, even for queries that contain very few attributes. Therefore, Hydra is not a viable option for summarizing streams that contain many attributes.

4.3 Evaluation of 𝕊1 with different streams

Our next set of experiments focused on investigating the effect of the stream properties (distribution of the attribute values and number of records in the stream) to 𝕊1's efficiency and accuracy.

Figure 7 shows the required time for summarizing 5 Million records from the following streams:[9]

- **Zipf,** $\alpha \in [1, 1.3, 1.5]$: Three different streams. Each of these streams contains 5 attributes (integer values). We generate the records by drawing random numbers from Zipfian distributions with exponent $\alpha \in [1, 1.3, 1.5]$.
- **Uniform**: The stream contains 5 integer attributes. The values for all attributes are drawn by a uniform distribution.
- **CAIDA and SNMP**: A vertical partitioning of the CAIDA and SNMP streams, containing 5 attributes.

We see that 𝕊1 requires less than 100 seconds to summarize each stream, even for the 200 MB quota. Also, the ingestion time for the Zipf streams is notably smaller compared to the uniform stream, and is further reduced as the Zipf's α value increases. This behavior can be traced back to the effort required for maintaining the K-minwise samples, which aligns with our earlier observation of the subtle elbow in Figure 3. As the Zipf exponent value rises in this experiment, the distribution of values becomes more skewed, causing more records to hash into the same cells at the corresponding attribute sketches. Consequently, the K-minwise samples of these cells swiftly reach an almost stable state. This pattern is also observed in the CAIDA and SNMP streams, where certain attributes also follow a Zipfian distribution, resulting in faster ingestion compared to the uniform stream. Still, even for the extreme case that the values for all stream attributes follow a uniform distribution, 𝕊1 throughput surpasses 150k updates per second.

The observed estimation error on these datasets is shown in Figure 6. The presented results correspond to the average error over 450 queries containing three predicates. We see that the observed error remains very low in all cases. For the case of Uniform, the average error is close to 0, whereas the highest error is observed with Zipf, $\alpha = 1.5$. The reason for this is because, as the Zipf exponent grows, the samples of the popular cells in the sketch (the cells responsible for storing the values with frequencies at the head of the Zipf distribution) end up containing almost exclusively records of these popular values. As a result, not sufficient samples remain for low-frequency predicates that happen to fall in the same popular cells, leading to higher estimation errors. This limitation is common across all small-memory sketches that provide error relevant to the stream size, e.g., the Count-min sketch [6].

Our final experiment with point queries aimed to explore the impact of the stream size to the performance of 𝕊1. Since we already demonstrated that the throughput of 𝕊1 is unaffected with the stream size (Figure 3), this experiment focused solely on the impact of the stream size to accuracy. Precisely, we used 𝕊1 to summarize CAIDA, which was the largest real dataset with 109 Million records. At regular intervals, we paused the stream ingestion, executed a fixed set of 3304 queries on the sketch, and computed the estimation error per query. The depicted results, shown in Figure 8, correspond to the mean observed error per interval. As predicted by the analysis, the observed error in OmniSketch stays stable with the stream size.

Summary. 𝕊1 is efficient in all cases, offering throughput that exceeds 150k updates per second. In terms of efficiency, the most difficult distribution is the uniform distribution, because it takes longer for the samples in the sketch cells to reach to a stable state. In terms of accuracy, Zipf distributions with extremely high values are more difficult for 𝕊1. This is a limitation shared across all small-memory sketches. Furthermore, increasing the stream size does not have a significant influence on the observed error.

4.4 Range queries

The final set of experiments was for evaluating the performance of 𝕊1 for range queries. The queries for this experiment were generated by choosing random p-dimensional ranges of length 2^{25}, for different values of p. In all cases, the starting and ending points of the ranges were within the minimum and maximum values of the corresponding attributes. In the following we report results for $p = \{2, 3\}$. While 𝕊1 (and its theoretical analysis) impose no constraints on the values of p and the query range length, higher p values or smaller query ranges always led to zero selectivity on our datasets – no records matched the queries. Even in these cases, 𝕊1

[8]The error at the Uniform stream was close to 0 and is thus omitted from the figure.
[9]Details for the generation of all streams are included in the project's github

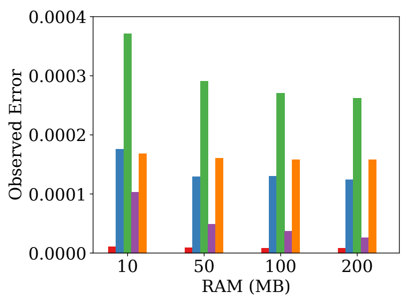

Figure 6: Observed error of 𝕊1 for different streams. Legend same as Fig. 7.[8]

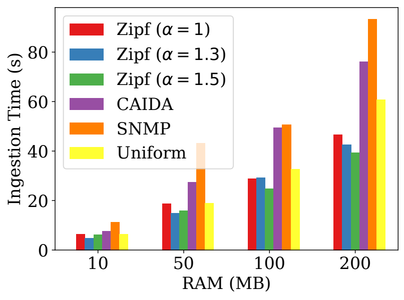

Figure 7: Ingestion time of 𝕊1 for different streams

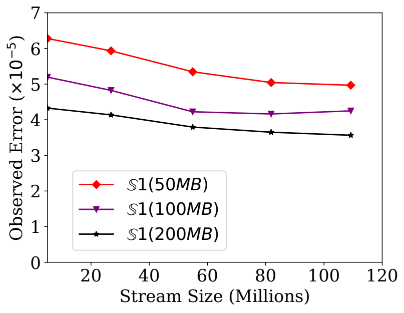

Figure 8: Estimation error of 𝕊1 for streams of different lengths.

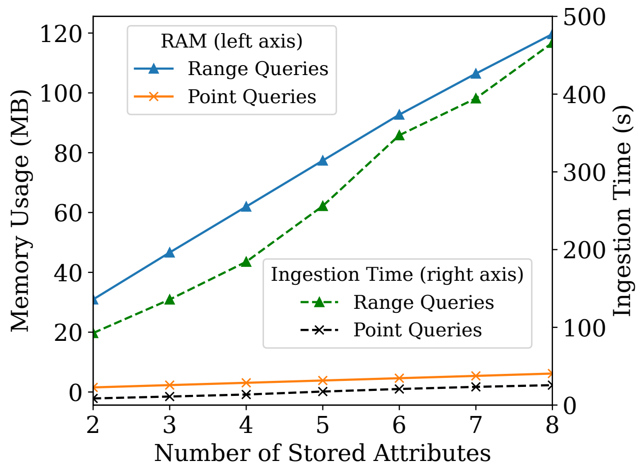

Figure 9: Memory and time needed to support range queries for 2 - 8 attributes.

still supports our theoretical guarantees for such queries, usually with an extremely small error. However, to be able to focus on more challenging configurations here, we do not present detailed results for higher values of p and smaller ranges. Also notice that not all stream attributes were suitable for range queries – some attributes were categorical (e.g., attribute `protocol` in the IP header).

Figure 9 shows the required time and memory for summarizing the SNMP stream (attributes ifOutOctets, ifOutErrors, ifOutDiscards, ifOutUcastPkts, ifInOctets, ifInUcastPkts, ifInErrors, ifInDiscards), with the range-enabled 𝕊1. The results correspond to $B = 1000$, $w = 20$, and $d = 3$. As an indication, the figure also includes the required time for summarizing the same stream with the standard 𝕊1 sketches that are not configured for supporting range queries. Similar to the standard 𝕊1, both memory and ingestion time increase linearly with the number of attributes. This is expected, since each new attribute requires a new attribute sketch. Also, the range-enabled 𝕊1 is slower and requires more memory for keeping the same number of samples. The reason for this performance degradation is that in the range-enabled sketch, the number of internal attributes sketches is increased by a logarithmic factor, for storing statistics for the dyadic ranges. Interestingly, though, the observed degradation is much smaller than the theoretical prediction. This discrepancy sources from the sketches responsible for storing the large dyadic ranges; recall that the number of large dyadic ranges is much less than the number of small dyadic ranges. For example, for an attribute with domain size 2^{32}, there exist only four dyadic ranges of size 2^{30}. As such, the cells storing large dyadic ranges are mostly empty, and require only a few bytes of RAM, whereas the few non-empty cells ($\approx 4 \times d$ per predicate) are still limited to B samples. This leads to both a faster ingestion (less frequent changes at the samples) and a smaller memory footprint. The mean observed error for range queries with $p = 2$ is 0.094 (886 queries) and for $p = 3$ is 0.072 (99 queries). These are within the bound shown in Section 3.3.

Summary. Our last set of experiments demonstrated that the range-enabled 𝕊1 maintains high performance and high accuracy also for range queries. Storage and computational complexity of maintaining 𝕊1 grows linearly with the number of attributes.

5 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.

REFERENCES

[1] 2011. The CAIDA UCSD Anonymized Internet Traces. https://www.caida.org/catalog/datasets/passive_dataset

[2] Noga Alon, Yossi Matias, and Mario Szegedy. 1999. The Space Complexity of Approximating the Frequency Moments. *J. Comput. Syst. Sci.* 58, 1 (1999), 137–147.

[3] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (jul 1970), 422–426.

[4] Moses Charikar, Kevin C. Chen, and Martin Farach-Colton. 2002. Finding Frequent Items in Data Streams. In *Automata, Languages and Programming, 29th International Colloquium, ICALP 2002 (Lecture Notes in Computer Science)*, Vol. 2380. Springer, 693–703.

[5] Graham Cormode, Minos Garofalakis, Peter J. Haas, and Christopher M. Jermaine. 2012. Synopses for Massive Data: Samples, Histograms,Wavelets, Sketches. *Foundations and Trends in Databases* 4, 1-3 (2012), 1–294.

[6] Graham Cormode and S. Muthukrishnan. 2005. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. *J. Algorithms* 55, 1 (apr 2005), 58–75.

[7] Anirban Dasgupta, Kevin J. Lang, Lee Rhodes, and Justin Thaler. 2016. A Framework for Estimating Stream Expression Cardinalities. In *ICDT (LIPIcs)*, Vol. 48. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:17.

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

[9] Philippe Flajolet and G. Nigel Martin. 1985. Probabilistic Counting Algorithms for Data Base Applications. *J. Comput. Syst. Sci.* 31, 2 (1985), 182–209.

[10] Sumit Ganguly, Minos Garofalakis, and Rajeev Rastogi. 2004. Tracking Set-Expression Cardinalities over Continuous Update Streams. *VLDB J.* 13, 4 (2004).

[11] Phillip B. Gibbons. 2001. Distinct Sampling for Highly-Accurate Answers to Distinct Values Queries and Event Reports. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*. Morgan Kaufmann, 541–550. http://www.vldb.org/conf/2001/P541.pdf

[12] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin J. Strauss. 2002. How to Summarize the Universe: Dynamic Maintenance of Quantiles. In *Proceedings of the 28th International Conference on Very Large Data Bases* (Hong Kong, China) *(VLDB '02)*. VLDB Endowment, 454–465.

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

[14] Jakub Lemiesz. 2021. On the Algebra of Data Sketches. *Proc. VLDB Endow.* 14, 9 (may 2021), 1655–1667.

[15] Jakub Lemiesz. 2023. Efficient Framework for Operating on Data Sketches. *Proc. VLDB Endow.* 16, 8 (jun 2023), 1967–1978.

[16] Ping Li and Arnd Christian König. 2011. Theory and Applications of B-Bit Minwise Hashing. *Commun. ACM* 54, 8 (aug 2011), 101–109.

[17] Ping Li and Christian König. 2010. B-Bit Minwise Hashing. In *Proceedings of the 19th International Conference on World Wide Web* (Raleigh, North Carolina, USA) *(WWW '10)*. Association for Computing Machinery, New York, NY, USA, 671–680.

[18] Ping Li, Art B. Owen, and Cun-Hui Zhang. 2012. One Permutation Hashing for Efficient Search and Learning. *CoRR* abs/1208.1259 (2012). arXiv:1208.1259 http://arxiv.org/abs/1208.1259

[19] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, Marinho P. Barcellos, Jon Crowcroft, Amin Vahdat, and Sachin Katti (Eds.). ACM, 101–114.

[20] Antonis Manousis, Zhuo Cheng, Ran Ben Basat, Zaoxing Liu, and Vyas Sekar. 2022. Enabling Efficient and General Subpopulation Analytics in Multidimensional Data Streams. *Proc. VLDB Endow.* 15, 11 (jul 2022), 3249–3262.

[21] S. Muthukrishnan. 2005. Data Streams: Algorithms and Applications. *Foundations and Trends in Theoretical Computer Science* 1, 2 (2005).

[22] Rasmus Pagh, Morten Stöckel, and David P. Woodruff. 2014. Is Min-Wise Hashing Optimal for Summarizing Set Intersection?. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Snowbird, Utah, USA) *(PODS '14)*. ACM, 109–120.

[23] Odysseas Papapetrou, Minos N. Garofalakis, and Antonios Deligiannakis. 2015. Sketching distributed sliding-window data streams. *VLDB J.* 24, 3 (2015), 345–368.