Adaptive Spatial Partitioning for Multidimensional Data Streams[1]

John Hershberger[2], Nisheeth Shrivastava[3], Subhash Suri[3], and Csaba D. Tóth[4]

Abstract

We propose a space-efficient scheme for summarizing multidimensional data streams. Our sketch can be used to solve spatial versions of several classical data stream queries efficiently. For instance, we can track ε-*hotspots*, which are congruent boxes containing at least an ε fraction of the stream, and maintain hierarchical heavy hitters in d dimensions. Our sketch can also be viewed as a multi-dimensional generalization of the ε-approximate quantile summary. The space complexity of our scheme is $O(\frac{1}{\varepsilon}\log R)$ if the points lie in the domain $[0, R]^d$, where d is assumed to be a constant. The scheme extends to the sliding window model with a $\log(\varepsilon n)$ factor increase in space, where n is the size of the sliding window. Our sketch can also be used to answer ε-approximate rectangular range queries over a stream of d-dimensional points.

1 Introduction

Many current and emerging technologies generate *data streams*, data feeds at very high rates that require continuous processing. There are many sources of such data: sensor networks monitoring environments such as industrial complexes, hazardous sites, or natural habitats; scientific instruments collecting astronomical, geological, ecological, or weather-related observation data; as well as measurement data for monitoring complex distributed systems such as the Internet. Despite their obvious differences, there are many similarities and commonalities among these data: they are all multidimensional; they have large volume; much of the data is routine and uninteresting to the user; and the user typically is interested in detecting unusual patterns or events.

As a result, there has been enormous interest in designing data-processing algorithms that work over a stream of data. These algorithms look at the data only once and in a fixed order (determined by the stream-arrival pattern). We assume that the algorithm has access to only a limited amount of memory, which is significantly smaller than the size of the stream seen so far, denoted by n. The algorithm must therefore represent the data in a concise "summary" that can fit in the available memory and can be used to provide guaranteed-quality approximate answers to user queries.

In the relatively short span of a few years, a substantial literature has developed on data stream processing. We refer the reader to two excellent surveys on data streams [4, 27]. Much of this work, however, has focused on single-dimensional data streams. In many natural settings, the true structure of the data is revealed only by considering multiple dimensions. For instance, IP packet data are multidimensional but, in order to simplify the analysis, packet data are often converted to a single-dimensional stream using "flow IDs." This puts too much burden on the network administrator who must "manually" decide which header fields to use to form flow IDs, especially when millions of flows

[1]Research by the last three authors was partially supported by National Science Foundation grants CCR-0049093 and IIS-0121562.

[2]Mentor Graphics Corp. 8005 SW Boeckman Road, Wilsonville, OR 97070, USA, and (by courtesy) University of California at Santa Barbara, john_hershberger@mentor.com.

[3]Department of Computer Science, University of California at Santa Barbara, Santa Barbara, CA 93106, USA, {nisheeth,suri}@cs.ucsb.edu.

[4]Department of Mathematics, Room 2-336, MIT, Cambridge, MA 02139, USA, toth@math.mit.edu.

may be passing through a single router. Instead, a flexible and programmable software tool that can automatically track the anomalous traffic pattern would be highly useful. In general, a summary of multidimensional point streams that can answer a variety of (statistical and aggregate) queries will be useful in many applications, and it is the focus of our paper.

Because of limited memory, exact answers for many problems are impossible to compute in the data stream model. There are many algorithms that can track "heavy hitters" or maintain approximate quantiles with error at most εn, using $O(\frac{1}{\varepsilon}\log \varepsilon n)$ memory. The simpler majority-finding algorithms (e.g., [21, 25]) maintain only a superset of the approximately heavy items, possibly including many false positives, and require a second pass to sort out the low frequency items.

We propose a novel, versatile, and space-efficient scheme for summarizing the spatial distribution of a d-dimensional point stream. Our *adaptive spatial partitioning* (ASP) scheme is fully deterministic and combines three basic data structures: a quadtree-like *ASP-tree* and two search trees. Specifically, given a stream of d-dimensional points $p_1, p_2, \ldots, p_n, \ldots$, drawn from the domain $[0, R]^d$, the ASP is an $O(\frac{1}{\varepsilon}\log R)$ size data structure, where ε is a user-specified parameter. It can answer a variety of geometric queries with a guaranteed error bound (and no low-frequency false positives).

All space bounds in this paper assume the *RAM* model of computation, in which each counter value can be stored in a single word. If the space is measured in bits, then the size of our data structure grows by a factor of $\log \varepsilon n$. Most of the related previous work [18, 22, 23] also suffers a similar increase in space under the bit complexity model.

1.1 Sample Applications

Let us consider a few applications of ASP before discussing further details.

Hot and cold spots. In many applications of multidimensional data streams, no single item occurs with large frequency; instead the users want to track small regions with many points (hot spots) or large regions with few points (cold spots). These are natural spatial versions of the well-studied problems of iceberg queries or heavy hitters in one dimension. Formally speaking, we define a *hot spot* to be a unit-size lattice box containing at least εn points. In the *hierarchical* version of hot spots, we want to track all εn-heavy lattice hypercubes where all εn-heavy descendants are *subtracted* from each hypercube.

A *cold spot* is a region of low stream density, which can also be useful in some applications. For example, cold spots can signal partial system failure in a distributed sensor network, or suggest useful data mining rules (e.g., in an insurance company database, a cold spot may characterize customer profiles that have a low claim rate.)

Rank and range queries. When dealing with multidimensional data, an important statistical concept is the *rank* of a point p, which is the number of points in the set that are *dominated by p on all coordinates* [6]. This is a natural spatial analog of single dimensional rank and quantile.

Related to rank computation is the problem of *multidimensional range queries*—given an axis-aligned rectangular box, determine the number of stream points that lie in the box. Many statistical analyses use range queries as a central tool. As one application, range queries over an IP packet stream can be used to estimate the traffic matrix over various source, destination, or application aggregates; for example, to determine the approximate P2P traffic between two campus sites. In sensor network data streams, range queries such as "what is the aggregate value of some signal in a certain region" are quite natural.

Thus, there are many spatial queries that arise naturally when dealing with streams of multidimensional points. Furthermore, many multidimensional data streams arise in monitoring applications, where users may not know *a priori* what questions to ask, and they may wish to build flexible and versatile

summaries that can handle many different queries. Our ASP provides one such solution—a single summary that can give guaranteed quality approximations to multiple fundamental geometric queries. The ASP is also very simple to implement.

Quantile summaries. While there are no similar multidimensional schemes for us to compare with ASP, it is interesting to compare the 1-dimensional version of the ASP with the best known schemes for quantile summaries [18, 23]. Because ASP is an arbitrary-dimensional[1] and general-purpose scheme, it is not optimized for rank or quantile approximation. However, in our empirical simulation, we observed that it offers acceptable performance for these queries. The ASP uses about twice as much space as the Greenwald–Khanna scheme [18], which is currently the most space-efficient scheme known for quantile summaries, but only about one-fourth to one-tenth the space of the Manku–Rajagopalan–Lindsay scheme [23].

Sliding windows and extensions. Our ASP scheme extends easily to the *sliding window* model of a data stream, with an $O(\log \varepsilon n)$ factor blowup in space, where n is the window size. It also works for a monotone version of the *turnstile model* and for *weighted* point streams, where each point has a (positive) weight, and the approximation error is measured as a fraction of the total weight. Further, two (or more) ASP structures constructed over separate data streams can be easily combined to form an aggregated summary over the union of streams, which makes it efficient for implementation in a distributed or parallel computing environment. This has already been used to compute approximate summaries in a sensor network setting [30].

1.2 Related Previous Work

One of the best-studied problems in data streams is finding *frequent items*; variants of this problem are also called *iceberg queries* [16] in databases and *heavy hitters* [15] in networking. The best deterministic stream algorithm for this problem is the *lossy counting* scheme of Manku and Motwani [22], which uses $O(\frac{1}{\varepsilon} \log \varepsilon n)$ space. (There are simpler schemes, using $O(1/\varepsilon)$ space, by Misra and Gries [25], Karp, Shenker, and Papadimitriou [21] and Demaine, López-Ortiz, and Munro [14], but they suffer from the false positive problem—they cannot guarantee any lower bound on the frequency of the items found.) Under the sliding window model, Golab et al. [13] proposed heuristics for maintaining hot items that is prone to return false negatives. Cormode et al. [9] consider a hierarchical version of the one-dimensional heavy hitter problem using $O(\frac{h}{\varepsilon} \log \varepsilon n)$ space, where h is the depth of the hierarchy.

Approximate quantile computation is another well-studied problem in data streams. For this problem, a classical result of Munro and Paterson [26] shows that any algorithm for computing *exact* quantiles in p passes requires $\Omega(n/p)$ space; thus, any single-pass, sublinear-space algorithm must resort to approximation. The best deterministic stream algorithm for ε-approximate quantiles is due to Greenwald and Khanna [18], and uses $O(\frac{1}{\varepsilon} \log \varepsilon n)$ space. Just recently, Arasu and Manku [3] have extended the Greenwald–Khanna algorithm to the sliding window model; their algorithm uses $O(\frac{1}{\varepsilon} \log \frac{1}{\varepsilon} \log n)$ space, where n is the window size.

Multidimensional stream processing is less well-developed, but several interesting problems have already been addressed. For instance, Cormode and Muthukrishnan [10], Agarwal, Har-Peled, and Varadarajan [1], and Hershberger and Suri [20] have proposed stream algorithms for extremal geometric structures, such as convex hull, diameter, width, etc. The challenge in maintaining such objects is that any data point can change the output dramatically and has to be accounted for; none of these schemes summarizes the *density distribution* of the stream. Charikar, O'Callaghan, and Panigrahy [8] have considered the k-median problem; Thaper et al. [32] have considered multidimensional histograms; both use randomized techniques. In [15], Estan, Savage, and Varghese propose the notion of hierarchically

[1]The space complexity of ASP depends on the dimension: the size of the d-dimensional ASP is $O(2^d \frac{1}{\varepsilon} \log R)$.

defined multidimensional *hotspots* for network monitoring. However, their solutions are offline, and so do not work in the streaming model.

Random sampling is an intuitively appealing idea for summarizing multidimensional point streams. Indeed, the *sticky sampling* scheme of Manku and Motwani [22] uses a random sample of size $O(\frac{1}{\varepsilon}\log \varepsilon n)$ to maintain single-dimensional heavy-hitters with high probability. If one wants high probability frequency estimates of any interval, then we need to use the concept of *ε-approximation*. A $\Theta(\frac{1}{\varepsilon^2}\log\frac{1}{\varepsilon})$ size random sample is an ε-approximation with high probability [33]. Unfortunately, in many important streaming applications, $\Theta(\frac{1}{\varepsilon^2}\log\frac{1}{\varepsilon})$ space is infeasible; for instance, it is not uncommon to use $\varepsilon = 10^{-4}$ or $\varepsilon = 10^{-6}$ in monitoring of IP packet streams. Many exact and approximate data structures are known for multidimensional range counting, but they all make multiple passes over the data (e.g., schemes based on Kd-trees, range trees [7], or PK-trees [35]). Likewise, ε-approximations are useful for range searching, but the traditional schemes for their construction require multiple passes over the data. In a forthcoming paper, Bagchi et al. [5] have proposed a deterministic streaming algorithm for constructing the ε-approximation. Unfortunately, besides the summary size of $\Theta(1/\varepsilon^2)$, which is already unattractive in many applications, their scheme requires a large amount of working space—$O(\varepsilon^{-(2d+2)})$ to be exact. A randomized algorithm of Manku, Rajagopalan, and Lindsay [23, 24] can maintain an ε-approximate ϕ-quantile for any fixed $\phi \in (0,1)$ in $O(\frac{1}{\varepsilon}\log^2\frac{1}{\varepsilon})$ space. A very recent randomized algorithm of Suri, Tóth, and Zhou [31] can maintain an ε-approximation of size $O(\frac{1}{\varepsilon}\log^{2d+1}\frac{1}{\varepsilon})$ for axis-aligned boxes in $\mathbb{R}^d$ with constant probability. This algorithm is, however, only of theoretical interest because it requires substantial working space (i.e., solving $O(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon}\log(\varepsilon n))$ linear programs in $O(\frac{1}{\varepsilon}\log^{2d+1}\frac{1}{\varepsilon})$ variables).

Under the *turnstile model*, in which both insertions and deletions are allowed, only randomized schemes are known and only for one-dimensional data. For approximate quantiles, the best scheme is the random ε-approximation of all inserted and all deleted items in $O(\frac{1}{\varepsilon^2}\log\frac{1}{\varepsilon})$ space (Gilbert et al. [17] proposed an alternative scheme that uses $O(\frac{1}{\varepsilon^2}\log^2 R\log\log R)$ space). Cormode and Muthukrishnan [11] can maintain ε-hot items in $O(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon})$ space provided that the joint frequency of all non-hot items is $O(\varepsilon n)$.

2 Adaptive Spatial Partitioning

In this section, we describe our summary scheme and establish its space and update complexity. We describe the scheme for 2-dimensional points for ease of exposition, but the ideas generalize readily to d dimensions, for any fixed $d \in \mathbb{N}$, with an $O(2^d)$ factor increase in space.

We assume that the input stream consists of points from the universe $[0,R]^2 \subset \mathbb{R}^2$. The point coordinates can have arbitrary precision, but our resolution is limited to 1; that is, only the most significant $\log R$ bits matter. (In some applications, such as network monitoring, the space is naturally discrete: $R = 2^{32}$ is the size of the IP address space. In other, non-discrete domains, such as astronomical or geological data, the resolution may be user-defined.) We make no assumption about the order in which points arrive. Without loss of generality, we assume that the i^{th} point arrives at time i. We denote the number of elements seen so far by n.

2.1 Data Structure

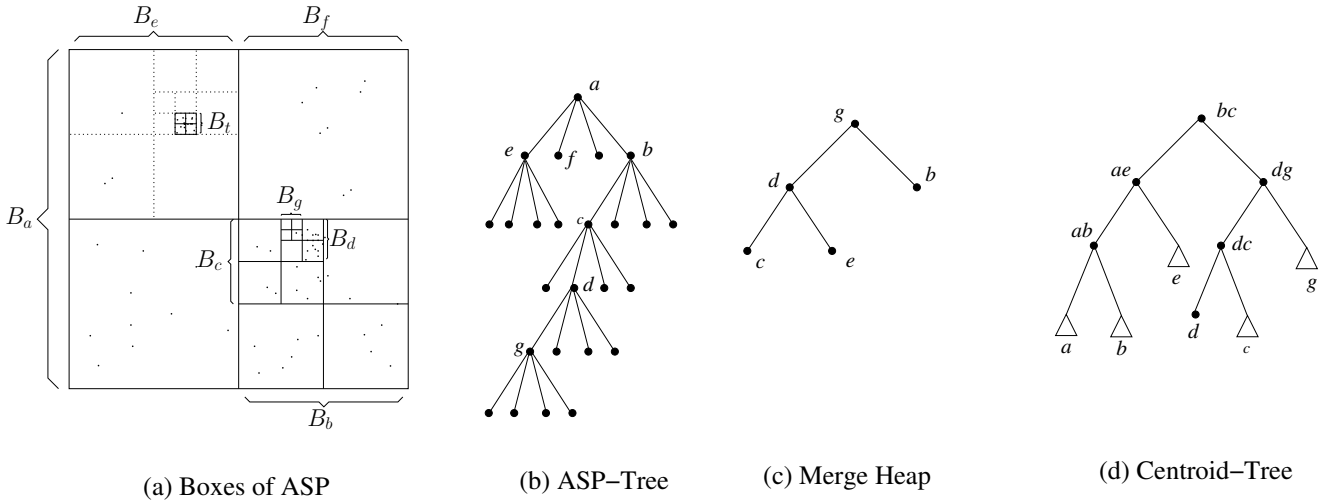
Our summary is based on a hierarchical decomposition of the space, which is represented in a data structure called the *adaptive spatial partitioning* tree. It is complemented by two auxiliary structures, a min-heap and a point location tree, that are used to perform merge and point location operations efficiently. We now describe the details of these data structures, which are illustrated in Figure 1.

The **adaptive spatial partitioning** tree (*ASP tree*) T is a 4-ary (in d dimensions, a 2^d-ary) tree. Each node $v \in T$ is associated with a grid-aligned region of the plane. We denote this region by B_v, and call it the *box associated with v*. We also maintain a counter for v, denoted $count(v)$. The box

associated with the root r of T is the entire universe $B_r = [0, R]^2$. The boxes follow the tree hierarchy: $u \in \mathit{children}(v)$ implies $B_u \subset B_v$ and the side length of B_u is at most half the side length of B_v. Because the maximum box size shrinks by at least half at every level, and the minimum resolution is one, the depth of the ASP tree is at most $\lceil \log R \rceil$. A given input point is contained in a sequence of $O(\log R)$ nested boxes corresponding to a path in T. However, each such point is associated with exactly one box B_v, and it is counted in $\mathit{count}(v)$ for the corresponding node v. The overall goal of the ASP scheme is to assign each input point to a leaf of the tree T, so long as the leaf contains at least εn points. When a point arrives, it is placed in the smallest active box containing it, i.e., it increments that box's counter. However, because the ASP tree is adaptive, the tree may change structure as points arrive, and boxes may be subdivided. This means that early-arriving points may be placed in an ancestor of the late-created leaf box that actually contains them.

The ASP tree is a compressed version of the standard quad-tree; in a quad-tree, the four children of a node have boxes with half the side length of the parent box, but in the ASP tree, though every node has four children, the boxes associated with the children may be much smaller. We maintain the invariant that the four children's boxes partition the box of v or one of its quad-tree descendants.

The ASP tree adapts to changes in the distribution of data over time. High density regions require a fine box subdivision, while for low density regions, a rougher subdivision is sufficient. For every point insertion, we increment n, the total stream size, and the counter of the smallest box B containing the new element. Thus every point is counted at exactly one node. If the counter of a node is above a *split threshold* αn, for a parameter α to be specified later, then we *refine* B by subdividing it into smaller boxes. Subsequently added points that fall in a child of B will increment $\mathit{count}(u)$ for some descendant B_u of B, but not the counter of B's node. Since n keeps increasing, the sum of the counts of a box B and its children may later be below the split threshold, and so the refinement is no longer necessary. We can *unrefine* B, by merging it with its children, to save memory. Because the children of B partition box B or one of its quad-tree descendants, we are allowed to unrefine only nodes with at most one non-leaf child. We call such nodes *mergeable*.



(a) Boxes of ASP (b) ASP–Tree (c) Merge Heap (d) Centroid–Tree

Figure 1: The boxes of the ASP for a distribution of input data points, the ASP tree, the corresponding merge heap, and centroid tree. The boxes in the ASP are drawn with solid lines. Dotted lines indicate boxes that were in the ASP tree at some time but have been removed by unrefinement operations. Box B_t was once the great grandchild of B_e, but in our current ASP tree, the children of t are stored as direct children of e. The nodes b, c, d, e, g are in the merge heap, since each has at least three leaf children.

While refinement is directly linked with the point being inserted, unrefinement is not. Naïvely, at

every point arrival, we should check all the mergeable nodes for possible unrefinement. Instead, we use a standard min-heap to maintain a priority queue of all the mergeable nodes; the top of the heap is the node that will become unrefinable earliest. We call this the **merge heap** M. The nodes in M are exactly the non-leaf nodes with *at least three children that are leaves in* T. The key of each node v is

$$key(v) \;=\; count(v) + \sum_{x \in children(v)} count(x).$$

The merge heap can be updated after an insert or delete in $O(\log |M|) = O(\log |T|)$ time, as described below.

The insertion of every point p requires searching in the ASP tree for the smallest box containing p. Using the ASP tree, the point location might take $O(\log R)$ time, the depth of the ASP tree. We can improve the time spent on point location among the $|T|$ nodes of the ASP tree to the optimal $O(\log |T|)$ per insertion with an **approximate centroid tree** C, a standard binary search tree defined on top of T. Every node $u \in C$ corresponds to a connected subtree T_u of T. If T_u has more than one node, then u stores an approximate centroid edge e_u of T_u that divides T_u into two roughly balanced parts; that is, each subtree must have size at least $c_d \cdot |T_u|$, for some constant c_d. (We can set $c_d = 2^{-d-1}$, in d-space, since the degree of every node is at most $2^d + 1$.) The two subtrees, in turn, correspond to the two children of u in C. Node $u \in C$ stores the edge e_u and the sizes of the two subtrees separated by e_u.

The query time to find the minimal box of T containing a point q is the depth of $C(T)$, which is $O(\log |T|)$ because the size of the subtrees decreases by a constant factor with the depth in C. Every non-leaf node $u \in C$ corresponds to an edge vw of T which, in turn, corresponds to the pair of boxes $B_w \subset B_v$ in $\mathbb{R}^d$. At node u, we test $q \in B_w$ and continue searching q in the left or right subtree of u accordingly. The search returns a leaf of C, which corresponds to a node $v(q) \in T$ and a box $B_{v(q)}$. The tests performed along the search prove that $q \in B_{v(q)}$, but $q \notin B_w$ for any $B_w \subset B_{v(q)}$, $w \in T$.

2.2 Maintaining the ASP

We initialize the ASP with a one-node ASP tree, with box B_r corresponding to the root, $count(r)$ set to zero, an empty merge heap, and a one-node approximate centroid tree C. We use a generic update parameter $0 < \alpha < 1$ in our discussion of the update operations for T and M. The parameter α is related to ε, but different applications in Section 3 will use different values of α.

When a new point p arrives, we find the smallest box in the ASP tree containing p and increment its count by one. This changes at most two keys in the merge heap (of the box and its parent), and we update the heap accordingly. For the majority of point insertions, this is all the work needed to update the ASP. Occasionally, however, we need to adjust the tree structures as well. We use two basic operations to maintain our summary, *refinement* and *unrefinement*, described below:

1. **Updates of the ASP tree.**

   Refinement. The *refinement* operation occurs when the counter of a node v exceeds a *split threshold* value αn. That is, if $count(v) > \alpha n$ and the box B_v is not a unit (minimum resolution) box, then we split B_v into four congruent boxes, and add edges from node v to the new children corresponding to the four half-size sub-boxes. (If v already had four children that did not cover B_v, then those children are detached from v and re-attached as children of one of v's four newly-created children.) As long as v's children cover B_v, each subsequent new point increments the counter of one of the descendants of v, and $count(v)$ does not change.

   Unrefinement. The *unrefinement* is performed when the key of a node in the merge heap is exceeded by the *merge threshold*. We set the merge threshold to be half of the split threshold, though this is tunable. Specifically, if v is a node in M and

$$key(v) \;<\; \frac{\alpha n}{2}, \tag{1}$$

Algorithm 1 ASP-Insert(Point p)

```
1: n ← n + 1;
2: v ← locate(p);
3: count(v) ← count(v) + 1;
4: Update v and parent(v) in M;
5: if count(v) > split threshold then
6:    Refine v;
7:    Update T and M; {see Section 2.2}
8: end if
9: while key(M.head) < merge threshold do
10:   Unrefine M.head;
11:   Update T and M; {see Section 2.2}
12: end while
```

then we merge the children of v into v, and set $count(v) := key(v)$. (Recall that $key(v)$ is defined as $count(v) + \sum_{x \in children(v)} count(x)$.) That is, we simply redirect the parent pointers of v's grandchildren to point to v, thereby deleting v's children, and fold the counters of v's children into v. It is worth noting that the updated v has at most four children, because no node in M can have more than four grandchildren.

2. **Updates of the Merge Heap.**

   Refinement. When we refine a node v, it automatically becomes mergeable, and is inserted into the merge heap. We then check if $parent(v)$ has more than two non-leaf children, in which case it is removed from the merge heap.

   Unrefinement. When the $key(v)$ of a mergeable node v drops below the merge threshold ($\frac{\alpha}{2}n$), then we unrefine v and delete it from the merge heap. After the unrefinement, the children and the keys of both v and its parent change; either node or both may need to be inserted or updated in the merge heap.

3. **Updates of the Centroid Tree.**

   Refinement. When a node v is refined in the ASP-tree, it is replaced by a 5-node subtree $R(v)$. In the approximate centroid tree C, we append to node v the centroid tree of $R(v)$, and at every ancestor of $v \in C$ we update the size of the subtree containing v.

   Unrefinement. In the ASP-tree, unrefinement means merging a subtree $R(v)$ of a node v_0 and its four children, three of which are leaves, into one node v. The unrefinement destroys four edges of T. In C, we delete four nodes corresponding to these edges: Every edge incident to a leaf in T corresponds to a leaf node in C. At this point, one of the remaining two edges incident to v_0 must correspond to a leaf node in C. We can perform the unrefinement by removing (recursively) four leaf nodes from C and updating the sizes of the subtrees at all their ancestors.

   Rebalancing. After adding or deleting four nodes, we do not necessarily rebalance the *entire* centroid tree. We use the partial rebuilding technique of Overmars [28]. If a subtree C_u of C becomes unbalanced (that is, if the approximate centroid e_u splits the tree T_u into two pieces, one of them smaller than $c_d|T_u|$), then we recompute the (exact) centroid tree of T_u. If multiple (necessarily nested) subtrees become unbalanced simultaneously, we rebalance the largest of them.

2.3 Complexity Analysis

We now prove that the space complexity of our data structures is $O(\frac{1}{\alpha})$, where α is the threshold parameter defined above. We also show that the amortized update time for point insertion is $O(\log \frac{1}{\alpha})$.

Theorem 2.1. *Given a split threshold α, the size of the ASP is $O(\frac{1}{\alpha})$, independent of the size of the data stream.*

Proof. We argue that the number of non-leaf nodes in T is $O(\frac{1}{\alpha})$. Since every node has at most four (2^d, in $\mathbb{R}^d$) children, this implies that the space complexity is $O(\frac{1}{\alpha})$. We first bound the number of nodes in the merge heap. No node in M satisfies the Merge Inequality (1)—otherwise, the unrefinement operation would have deleted that node from M. Thus, for each $v \in M$, we must have $key(v) \geq \frac{\alpha n}{2}$. The value $count(w)$ of a node $w \in T$ contributes to at most two keys, and so we have the following inequality:

$$|M| \cdot \frac{\alpha n}{2} \leq \sum_{v \in M} key(v) < \sum_{v \in T} key(v) = \sum_{v \in T} \left(count(v) + \sum_{x \in children(v)} count(x) \right) \leq 2 \sum_{w \in T} count(w) = 2n.$$

Thus the merge heap has at most $2n/(\frac{\alpha n}{2}) = 4/\alpha$ nodes.

Any non-leaf node that is not in M has at least two children that are also non-leaf nodes of T. These are branching nodes of the tree on paths to nodes in the merge heap. This implies that the total number of non-leaf nodes in T is at most $8/\alpha$. This completes the proof. □

We show next that the amortized updates require $O(\log \frac{1}{\alpha})$ time.

Theorem 2.2. *Given a split threshold α, the amortized time to update the ASP summary after a point insertion is $O(\log \frac{1}{\alpha})$, assuming that the data stream has size at least $n = \Omega(\frac{1}{\alpha})$.*

Proof. When a new point p arrives, we find the smallest box in the ASP tree containing p in $O(\log |T|) = O(\log \frac{1}{\alpha})$ time using the point location tree. After the insertion, we perform at most one refinement, which requires $O(1)$ time in the ASP tree, and update M, which requires $O(\log |M|) = O(\log \frac{1}{\alpha})$ time.

We show that we can also update the approximate centroid tree in amortized $O(\log \frac{1}{\alpha})$ time. (Using an analogous argument, Schwarz, Smid, and Snoeyink [29] showed that insert-only updates of a centroid tree C can be done in amortized $O(\log |C|)$ time.) Suppose e is the exact centroid edge of a tree T_u. Then e remains an approximate centroid as long as fewer than $\kappa|T_u|$ refinements and unrefinements modify T_u, for some constant κ. After $\Omega(|T_u|)$ events, it may be necessary to recompute the exact centroid tree of T_u, which takes $O(|T_u|)$ time [19]. Thus, rebalancing the approximate centroid tree requires $O(k)$ time after k refinements or unrefinements at each of the $O(\log |T|) = O(\log \frac{1}{\alpha})$ levels of C. Hence the amortized cost of the rebalancing is $O(\log \frac{1}{\alpha})$.

To bound the total number of unrefinements, we note that each refinement increases the number of nodes in T by 4, and each unrefinement decreases the number of nodes by the same amount. The total number of nodes is positive, and so there can be no more unrefinements than refinements. Like a refinement, an unrefinement takes $O(\log \frac{1}{\alpha})$ amortized time, and so the proof is complete. □

In our analysis, the amortized update time is dominated by the $O(\log \frac{1}{\alpha})$ point location query among the $O(1/\alpha)$ nodes. Since the updates of the ASP and the auxiliary search trees can also be done in amortized $O(\log \frac{1}{\alpha})$ time, we may assume that *every* point insertion causes changes to the ASP. In fact, a more careful analysis shows that the total number of refinements and unrefinements is much smaller, at most $O(\frac{1}{\alpha} \log(\alpha n))$. We use the following observation to bound the total number of unrefinements performed up to time n:

If $v \in T$ is refined at time n_0 and unrefined at a later time $n_t > n_0$, then we must have $n_t > 2n_0$, because the keys increase monotonically. That is, from the moment of refinement, the stream size must at least double before v becomes eligible for unrefinement.

We divide the total time $[1, n]$ into exponentially increasing time slices. Let

$$P_0 = \left[1, \frac{1}{\alpha}\right], \text{ and } P_i = \left[\frac{2^{i-1}}{\alpha} + 1, \frac{2^i}{\alpha}\right], \quad \text{for} \quad i = 1, 2, \ldots, \lceil \log(\alpha n) \rceil.$$

In the interval P_0, there are only $\frac{1}{\alpha}$ input points; we create at most $O(\frac{1}{\alpha})$ nodes, and so the number of unrefinements cannot be more than $O(\frac{1}{\alpha})$. In an interval P_i, $i > 0$, every unrefined node must have already existed at time $n = 2^{i-1}/\alpha + 1$. But there are only $O(\frac{1}{\alpha})$ such nodes, and so the number of unrefinements is also $O(\frac{1}{\alpha})$. The total number of merge operations over the period $[1, n]$ is $O(\frac{1}{\alpha} \log(\alpha n))$. Since a refinement increases the number of nodes by four while an unrefinement deletes four nodes, and the size of the ASP is always $O(\frac{1}{\alpha})$, the number of refinements is also bounded by $O(\frac{1}{\alpha} \log(\alpha n))$.

3 Applications

Our approximations will have an absolute error of εn. Just as for iceberg queries and quantile summaries, this error guarantee is the best possible for schemes using roughly $O(1/\varepsilon)$ space. Specifically, Alon, Matias, and Szegedy [2] prove that, given a data stream in a domain of size R, any (randomized or deterministic) algorithm that approximates the frequency of the most frequent item to within a constant *relative* error must use $\Omega(R)$ space. Because hot spots and range counting problems are strict generalizations of iceberg and quantile summaries, these approximation quality lower bounds apply to ASP as well.

3.1 Hot and Cold Spots.

Hot Spots. We define a *hot spot* in $\mathbb{R}^d$ as an integer unit cube containing at least εn points of the stream. Hot spots are a natural spatial analogue of iceberg queries: in multidimensional data streams, none of the individual points may occur with large frequency, yet a closely clustered set of points may signal unusual or suspicious activity. Formally, an *integer unit cube* is defined as $\prod_{i=1}^{d} [x_i, x_i + 1)$ where each x_i is an integer in the range $\{0, 1, \ldots, R - 1\}$. Hypercubes of larger granularity or more general rectangular templates are easy to handle as well.

To compute ε-hot spots, we maintain a d-dimensional ASP with parameter $\alpha = \frac{\varepsilon}{2 \log R}$, and report every leaf node that corresponds to a unit cube and whose counter exceeds $\frac{\varepsilon}{2} n$.

We store the counters of all hot spots in a min-heap (of size $O(\frac{1}{\varepsilon})$) so that we can maintain a list of hot spots at all times by inserting and removing cubes whose frequency reaches or drops below $\frac{\varepsilon}{2} n$. Observe that, for each reported hot spot v, points that fell in v before it was created were added to counters of ancestors of v. Thus, the true count for v is at most $\mathit{count}(v) + \sum_{u \in \mathit{ancestor}(v)} \mathit{count}(u) < \mathit{count}(v) + \lceil \alpha n \rceil \log R = \mathit{count}(v) + \lceil \varepsilon n / (2 \log R) \rceil \log R = \mathit{count}(v) + \frac{\varepsilon}{2} n + \log R$. This proves that the maximum possible error in our estimate is $\frac{\varepsilon}{2} n$ and it is easy to see that *ASP reports all the ε-hot spots.* On the other hand, every reported cube B_v contains at least $\mathit{count}(v) \geq \frac{\varepsilon}{2} n$ points, and hence *ASP reports no low-frequency false positives.*

The approximation guarantee of the ASP can be improved at the expense of increased space and time. For instance, by choosing $\alpha = \frac{\varepsilon \delta}{\log R}$, for any $\delta \in (0, 1)$, we can guarantee that each reported hot spot contains at least $(1 - \delta)\varepsilon n$ points. By choosing the hot spots as *unit* cubes, we have exploited the maximum power of our summary. One can also extract hypercube regions of larger granularity from our data structure.

Cold Spots. The εn approximation error is unavoidable for any box in the hierarchy, just as it is for hot spots. For instance, any algorithm that finds the *largest empty box* in an $R \times R$ domain requires $\Omega(\min\{n, R^2\})$ space. A natural definition of the cold spot is the *largest* box with at most εn points. Since the leaf cells of our ASP are precisely the boxes with fewer than εn points, we can track the cold spot by, say, keeping these boxes sorted in descending order of their size-to-population ratios.

Hierarchical Hot Spots. Hierarchical hot spots are a spatial generalization of the *heavy hitter hierarchy*, introduced by Cormode et al. [9] for 1-dimensional data. In a rooted tree representation of the data, the hierarchical heavy hitters are defined recursively in a bottom-up order: each node whose subtree has frequency more than εn, not counting the frequencies of its heavy hitter descendants, is a heavy hitter.

In order to maintain hierarchical hot spots using ASP, we add a counter $des(v)$ at each node v to track the total population of all the descendants of v. That is, $des(v) = \sum_{w \in descendants(v)} count(w)$. This increases the worst-case per-point update time of the ASP from $O(\log \frac{1}{\alpha})$ to $O(\log R)$, which is the depth of T. To determine the heavy hitters at each level of T, we set $\alpha = \frac{\varepsilon}{2 \log R}$ and report every node of T such that the value $des(v) - \sum\{des(w) : w \text{ is a descendant of } v \text{ and a heavy hitter}\}$ is above $\frac{\varepsilon}{2}n$. The algorithm by Cormode et al. [9] solves this problem using $O(\frac{1}{\varepsilon} \log R \log(\varepsilon n))$ memory; in comparison, we can compute hierarchical heavy hitters according to the same definition using only $O(\frac{1}{\varepsilon} \log R)$ memory.

Most Frequent Items The *most frequent items* problem is a special case of 1-dimensional hot spots, and so the 1-dimensional ASP gives an alternative deterministic scheme for finding all ε-hot items, without low-frequency false positives. Specifically, with $O(\frac{1}{\varepsilon} \log R)$ space, we can report $O(1/\varepsilon)$ items that include *all* the hot items and have individual frequencies at least $\frac{\varepsilon n}{2}$. Our scheme is comparable to the best deterministic scheme for this problem, which is by Manku and Motwani [22] and uses $O(\frac{1}{\varepsilon} \log(\varepsilon n))$ space.

3.2 Rank and Range Queries.

Ranks and Quantiles. In a set of d-dimensional points, the *rank* of a point p is the number of points dominated by p on all coordinates. This is a natural generalization of the rank in one dimension, where the rank of an element x is the number of elements less than x. The rank and quantiles are intimately related: quantiles are elements of specified ranks; on the other hand, given the quantiles, the rank of an element can be located between two consecutive quantiles. Thus, the 1-dimensional ASP with $\alpha = \frac{\varepsilon}{\log R}$ can also be interpreted as an *ε-approximate quantile summary*, and thus offers an alternative deterministic scheme for this well-studied problem.

In one dimension, the boxes of the ASP tree T are intervals. We scan these intervals in sorted order of their right endpoints (by a simple traversal of T), and report the smallest integer $a(q)$ for which the sum of the counters of intervals in T whose right endpoint is at most $a(q)$ exceeds qn. (Note that each *unit* interval B_u contains only multiple copies of a single integer, which is either strictly smaller than q and counted correctly, or equals q and so it has no effect on the rank computation.) The error of our approximation is the sum of the counters of all non-unit intervals of T containing q in their interior, which is bounded by $(\log R - 1)\alpha n \leq \varepsilon n$. Conversely, given an element x, we can find its approximate rank with a maximal error of εn by reporting the rank of the left endpoint of the smallest interval in the ASP tree containing x.

Multidimensional Rank and Range Counting Determining the rank in more than one dimension is equivalent to *range counting* for *grounded* query rectangles, which are defined as $\prod_{i=1}^{d}[0, b_i]$. Furthermore, it can be used to perform range counting for general axis-parallel rectangles, because every such rectangle can be expressed with 2^d grounded rectangles using inclusion-exclusion. We can use the ASP

directly for multi-dimensional range counting with the following simple algorithm: Given a query box Q, we report the sum of all counters $count(v)$ weighted with the fraction of the volume of each box B_v lying in the query box, that is, $\sum_v count(v) \cdot \text{vol}(B_v \cap Q)/\text{vol}(B_v)$. Although the theoretical approximation quality of this simple scheme can be arbitrarily bad in the worst case, our experiments show that it performs well in practice.

With a recursive construction, we can use ASP to provide good theoretical guarantees for multidimensional range queries; however, the space bound for the construction grows as $O(\frac{1}{\varepsilon}\log^{2d-1} R)$. This is comparable with the best known *randomized* axis-aligned range counting algorithm [31], which maintains a summary of size $O(\frac{1}{\varepsilon}\log^{2d+1}\frac{1}{\varepsilon})$ but uses an advanced merging scheme.

We show how to use the ASP for grounded range counting in two-dimensions; the ideas extend readily to d dimensions, for any fixed $d \geq 2$. We maintain a one-dimensional ASP tree T based on the x-coordinates of the points, with $\alpha = \frac{\varepsilon}{2\log R}$. At each node $v \in T$, we maintain another one-dimensional ASP tree based on the y-coordinates of all the points assigned to v or its descendants. These secondary trees use $\alpha' = \frac{\varepsilon}{2\log^2 R}$. The total number of points represented at each level of T is at most n and there are at most $\log R$ levels. So the size of our data structure is $O(\frac{1}{\alpha'}\log R) = O(\frac{1}{\varepsilon}\log^3 R)$.

Given a grounded query rectangle $B = [0, b_1] \times [0, b_2]$, we first determine the set T_1 of maximal nodes $v \in T$ whose interval B_v covers $[0, b_1]$. There is at most one such node at each level, so $|T_1| \leq \log R$. We use the ASP of each node in T_1 to determine the rank of b_2. We report the sum of the ranks of b_2 in all those ASPs.

Clearly, the output is a lower bound of the number of points in B. The error is bounded by the sum of the errors of the quantile queries in the two dimensions: for the x-coordinate, the sum of counters of non-unit intervals of T that contain b_1, and for y-coordinate, the sum of counters of non-unit intervals in the ASP trees of T_1 that contain b_2. The x-error is bounded by $\alpha n \log R = \frac{\varepsilon}{2}n$. The y-error is at most $\sum_{j=1}^{\log R}(\alpha' n_j \log R) < \log R \cdot (\alpha' n \log R) = \frac{\varepsilon}{2}n$, and so the total error is no more than εn.

The technique generalizes to d dimensions, with a $\log^2 R$ factor increase in space for each additional dimension. Because the space requirement for range searching degrades rapidly with d, this scheme is suitable only for small dimensions, say, two or three.

4 Extensions of the Streaming Model

4.1 The Sliding Window Model

In some applications, the data stream is infinite, but only the "recent" portion of the stream is relevant. For instance, in network monitoring, routers continuously log packet information, but a network manager is typically interested in the data of the last few minutes, hours, or days. In financial data streams, the fresh data is of the highest significance. The same holds for many sensor-based monitoring systems as well. The *sliding window* model of data streams is motivated by these applications [4, 12, 27]. Let n denote the size of the window; that is, we want to maintain a summary over the last n points. A point p_i that is inserted at time i is deleted at time $i + n$, which we refer to as its *expiration time*. We now describe how to extend our summary technique to answer *sliding window* queries.

4.1.1 ASP tree in the Sliding Window Model

The ASP tree structure as described in Section 2 handles insertions only. If we were to extend our scheme to the sliding window model naïvely, then every time a point p expires, the counter where p was counted should be decremented. However, this is infeasible because our constant-size summary cannot maintain specific information about every point p and its expiration time. We use a variant of the *exponential histogram* [12] to maintain a more refined view of each counter $count(v)$ associated with a node v of the ASP tree.

In particular, we associate an exponential histogram EH(v) with every node v. The points contributing to EH(v) are partitioned among $O(\log n_v)$ buckets, where n_v denotes the number of points within the current window that are counted in EH(v). Each bucket tracks consecutive (in time) elements inserted into the box B_v. Each bucket has an *expiration time*, which is the expiration time of its youngest point. We store only the size and the expiration time of each bucket.

The sizes of the buckets are powers of two, and the bucket sizes increase with their age. There are at most two buckets of any given size, and whenever a bucket of size 2^i exists, there is also at least one bucket of size 2^{i-1}. We can approximate n_v using these bucket sizes, because only the oldest bucket has any portion that extends beyond the current window.

We can increment EH(v) by inserting the most recent point into a one-element bucket; whenever there are *three* buckets of equal size, we merge the two oldest into one new bucket. This ensures that the bucket sizes are exponentially increasing. The size of the oldest bucket (containing the oldest points) is at most $(n_v + 1)/2$. When the expiration time of a bucket is reached, the bucket is deleted and the size of EH(v) is decremented by the bucket's size.

Unfortunately, exponential histograms are incompatible with our unrefinement operation (refinement is straightforward, since we create new nodes while leaving the old counters intact). Unrefinement requires merging the contents of $2^d + 1$ independent counters. To avoid this operation, we modify our counter representation. We represent each $count(v)$ as a collection of exponential histograms, one of which is the histogram EH(v) associated with v; the others are the histograms of descendants of v that have been merged into v. Of course, we delete a histogram from $count(v)$ if all its buckets have expired.

Each counter of the original ASP scheme is now maintained as a collection of EHs, each containing a logarithmic set of bucket counters that maintain each bucket's size and expiration time.

As in the algorithm maintaining our ASP summary under the streaming model (Section 2), a counter $count(v)$ is modified by only three events: When a point arrives in the box B_v then EH(v) is incremented (and therefore also $count(v)$); at unrefinement five ($2^d + 1$, in $\mathbb{R}^d$) nodes are merged and the EHs in their counters are grouped into a single collection; and when a bucket in EH(v) expires then $count(v)$ decreases correspondingly.

4.1.2 Space and Time Complexity Analysis

Since we keep all the exponential histograms stored in the counter of v and its children after v is unrefined, the exponential histogram EH(v) might exist long after the node v is deleted from T. To prove the space bound for the ASP-tree, our primary goal is to show that at most $O(\frac{1}{\alpha})$ EHs coexist at any given time t.

If EH(v) is alive at time t, then it was incremented in the window $(t - n, t]$, and therefore node v existed at some point in this time range. We know from the analysis of the standard ASP tree that at time $t - n$ there are $O(\frac{1}{\alpha})$ nodes in T. The refinement of these nodes can generate $O(\frac{1}{\alpha})$ children. If a node that was not alive at time $t - n$ is refined, then it must receive at least αn points during the period $(t - n, t]$. There are at most $\frac{1}{\alpha}$ such nodes, as each of the last n points can contribute to the counter of at most one node. This shows that at most $O(\frac{1}{\alpha})$ nodes exist during $(t - n, t]$.

Since we create a new EH every time a node v is created, we also have to ensure that a node is not created (and then merged again into its parent) too many times in $(t - n, t]$. We show that every node $v \in T$ is refined at most twice within a window of size n.

> Assume that there are times $t_1 < t_2 < t_3 < t_4 < t_5$ such that node $v \in T$ is refined at t_1, t_3, and t_5 (i.e., $\alpha n < count(v) \leq \alpha n + 1$); and unrefined at t_2 and t_4 (i.e., $count(v) \leq \frac{\alpha}{2} n$). Thus, $count(v)$ loses the oldest $\frac{\alpha}{2} n + 1$ points in $(t_1, t_2]$ and then in $(t_3, t_4]$. Therefore the first point inserted in $[t_1, t_5)$ expires within the same interval, and so $n < t_5 - t_1$.

As noted earlier, EH(v) consists of $\log n_v$ buckets, so the full data structure requires $O(\sum_v \log |\text{EH}(v)|)$ space, where $|\text{EH}(v)|$ denotes the total number of points in the buckets of EH(v). The total number

of EHs is $O(\frac{1}{\alpha})$ at any time t, and $\sum_v |\mathrm{EH}(v)| \leq 2n$, so $O(\sum_v \log n_v) \leq O(\frac{1}{\alpha}\log(\alpha n))$. Hence for the sliding window model, ASP requires $O(\frac{1}{\alpha}\log(\alpha n))$ space.

The amortized time complexity of insertions into each $EH(v)$ is $O(1)$. The deletion of expired buckets and decrementing the corresponding counters likewise takes $O(1)$ time. However, point location and updating the centroid tree and merge heap still take $O(\log \frac{1}{\alpha})$ time. Thus, the per point processing time is dominated by point location, which requires $O(\log \frac{1}{\alpha})$ time.

4.1.3 Approximation quality

We note that from every exponential histogram, only the oldest bucket contains points outside the sliding window. As long as a bucket is alive it contains at least one point within the current window. This implies that the number of elements assigned to v within the window is approximated by $\frac{1}{2} \cdot |\mathrm{EH}(v)| < n_v \leq |\mathrm{EH}(v)|$.

We can improve the approximation quality (reduce the threshold for false positives) by a more accurate exponential histogram: Instead of at most two buckets of each power of two, we can have $\frac{1}{\delta}$ buckets of the same size. This reduces the error coming from the size of the oldest bucket from a factor of $\frac{1}{2}$ to $(1-\delta)$.

4.2 Turnstile Model

In the turnstile model [27], elements can be inserted and deleted arbitrarily (the order of deletions is independent of insertions). Each new entry of the stream either inserts a new element or deletes a previously inserted element. Such a model is well-suited for some applications, such as accounting or database transactions.

A single delete operation by itself is easy to perform: We find the smallest box containing the deleted element and decrement its counter by one. Unfortunately, over time, the errors can become arbitrarily large. The trouble is that the total number of points can fluctuate significantly. For example, suppose αn_1 points are represented by one box B when n_1 is large. But, due to deletions, the total number of points may later drop to $n_2 = \delta \cdot n_1$, where $\delta \in (0,1)$, and our ASP tree has no recollection of the spatial distribution of the $\frac{1}{\delta} \cdot \varepsilon n_2$ points in B.

In order to guarantee bounds on the error of approximation under the turnstile model, we restrict the *relative number* of deletions. Specifically, we assume that, at any time, the total number of deletions is at most half (or a constant fraction) of the total number of inserts. (In many applications, such as accounting or database systems, where deletions correspond to transaction cancellations, this restriction is quite realistic.) Our ASP tree can easily be adapted to this *monotone* turnstile model: We construct the ASP with parameter $\varepsilon' = \varepsilon/2$ and perform insertions as before; when deleting a point p, we locate the smallest box B containing p with strictly positive counter and decrement its count by one. The error in our structure is at most $\varepsilon n/2$, where n is the total number of insertions. Due to the monotonicity assumption, the number of (undeleted) elements is not less than $n/2$, hence the absolute error in the frequency of a box is at most an ε fraction of the total population. Our space analysis for the streaming model still remains valid.

4.3 Weighted Streams

In some applications, points have weights associated with them. For instance, in the networking application, packets have different sizes; in online banking, different transactions have different monetary value. All the problems mentioned earlier (hot spots, range counting, etc.) have natural weighted versions, and our ASP scheme extends easily to this generalized setting.

We assume that each element of the stream is a pair (p_i, w_i), where $p_i \in \mathbb{R}^d$ is the location of the point and $w_i \in \mathbb{N}$ is its weight. When maintaining the weighted ASP under the streaming data model,

we can increment the smallest box B_v containing p_i if $count(v) + w_1 \leq \alpha n$, otherwise we refine v and insert w_i into a new counter. (In the rare case of a large weight, that is, if $w_i > \alpha n$, we refine v until the unit box level, where a weight exceeding αn is allowed, and unrefine most of the newly created empty boxes above p_i).

Under the sliding window model, it is not obvious how to increment an exponential histogram by w_i. We can, again, do better than simply inserting w_i unit weight copies of the point p_i. We observe that if w_i copies of the point p_i are inserted, then they are stored in the $\log w_i$ smallest buckets. We wish to store them in a single bucket, since they all expire simultaneously. We modify the definition of the exponential histogram such that buckets of nominal size 2^i are allowed to have any size in the range $[2^i, 2^{i+1} - 1]$. Then when a pair (p_i, w_i) arrives, we store all w_i copies of p_i in a single bucket, and merge into this bucket the maximum number of small buckets $1, 2, \ldots$ such that the total weight is less than $2w_i$. Buckets above this one are merged as necessary to preserve the monotonicity and exponential relationship of bucket sizes. Thus the smallest $\Theta(\log w_i)$ buckets are removed. If the average weight of every point is large, this saves a significant amount of space. It remains true that at most half of the elements in the buckets can lie beyond the endpoint of the window even if $\log w_i$ is the only bucket, because at least half the points in this bucket correspond to the same expiration time.

5 Experimental Results

We implemented adaptive spatial partitioning for both single- and multi-dimensional streams. In this section, we briefly report on some of our experimental results.

For two-dimensional adaptive spatial partitioning, we used three different data sets: *random*, which consisted of n points generated uniformly at random inside the box $[0, R]^2$; *k-peaks*, which included k Gaussian distributions concentrated around k randomly chosen centers, against a background of low density uniformly scattered points (noise); and the *gazetteer* data from UCSB's Alexandria Digital Library (see Figure 2), which consists of the locations of 10^7 geographical features. In each case we used $R = 2^{20}$.

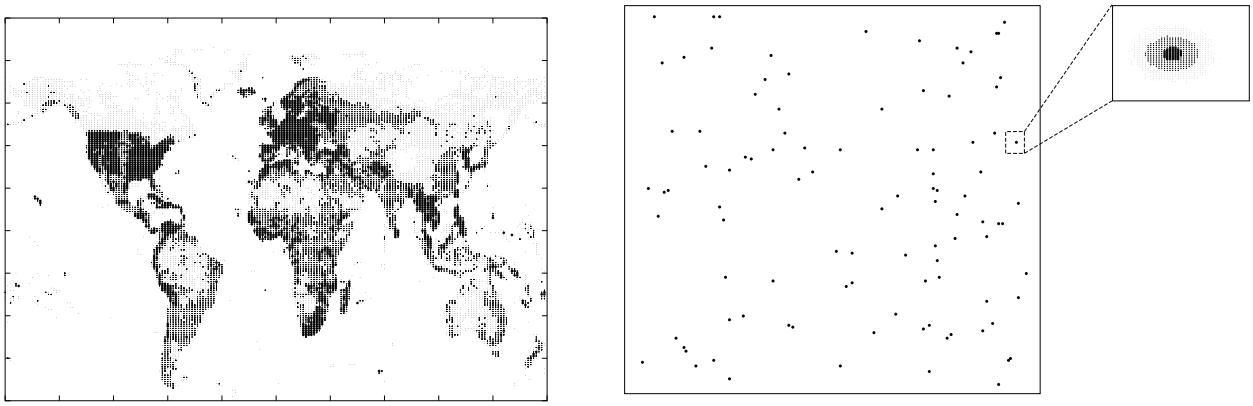


Figure 2: Input datasets: *gazetteer* (left) and *k-peaks* with $k = 100$ (right). The gray scale reflects the density of data points in each region. The inset picture on the right figure shows a Gaussian distribution at a single peak.

5.1 Memory Usage of ASP

	# points	ASP tree size $\varepsilon = 0.01$	0.001	0.0001
100-peaks	10^6	2269	3433	9457
	10^7	2237	3265	4229
	10^8	2237	3261	4061
random	10^6	5461	83093	492269
	10^7	5461	87381	387749
	10^8	5461	87381	349525
gazetteer	10^6	5965	56169	532933
	10^7	5853	55989	530585

Table 1: ASP tree size for different experimental data.

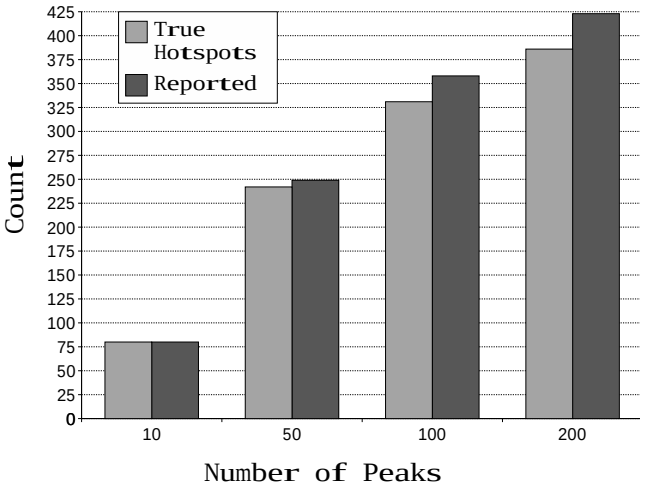


Figure 3: True vs. reported hot spots

Table 1 shows the space needed for the ASP as a function of n and ε. The main point to note is that, for a fixed value of ε, the size of the ASP is essentially independent of n. It increases linearly with $\frac{1}{\varepsilon}$, as predicted by the theoretical analysis. For the k-peaks data the size depends more on k than ε, because the ASP quickly "zooms" into the k-centers and maintains partitions only for the regions around them. Interestingly, for some data sets, the memory usage actually decreases as n increases. Our hypothesis is that this occurs because a larger fraction of nodes get unrefined near the top of the ASP tree. In the k-peaks dataset, for example, the counts of nodes corresponding to the region around peaks becomes quite heavy, while the counts of intermediate nodes in the path connecting these heavy nodes to the root will not increase. As size of the input stream increase these intermediate nodes will be slowly unrefined, thereby decreasing the total size of ASP.

5.2 Detecting Hot Spots with ASP

Figure 3 shows how well the ASP detects hot spots in the k-peaks data. In our experiment, a total of $n = 10^6$ points were streamed, and we fixed $\varepsilon = 10^{-3}$. Given a specific value of k, $10^6/(k+1)$ points were generated in each of the k Gaussian clusters, and the final $10^6/(k+1)$ points were distributed uniformly inside the square (as background noise).

In this experiment, we compared the number of hot spots reported by our ASP vs. the actual number of hot spots (see Figure 3).[2] Our algorithm detects all the true hot spots, and a small number of false positives as well. The false positives are within a factor of two of being hot. We also note (data not shown) that the size of the ASP tree grows essentially linearly with k, assuming that $\varepsilon < 1/k$, as is the case in these experiments.

5.3 Range Query

Our third experiment was designed to test ASP's use for range counting. We implemented the simple approximation scheme based on the ASP only (we report the sum of counters weighted with the overlapping areas $\sum_v count(v) \cdot \text{area}(B_v \cap Q)/\text{area}(B_v)$), rather than the multi-level data structure discussed

[2]We reduced the number of false-positive hot spots reported by an easy optimization: The maximum error E in an ASP tree is the maximum weight of any path to a leaf. In these experiments, we reported boxes having more than $\varepsilon n - E$ points as hot spots. This optimization reduces the number of false positives, but does not introduce any false negatives.

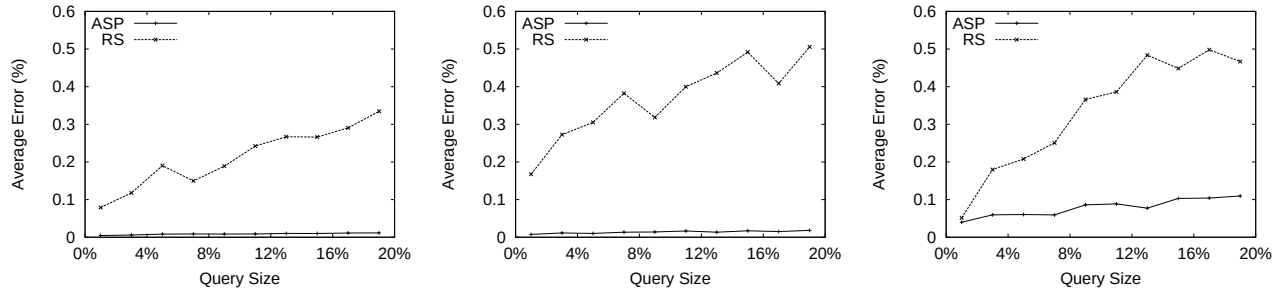


Figure 4: Approximation error in range queries: The figure are for (left to right) *random*, *k-peaks*, and *gazetteer* datasets. ($n = 10^6$ and $\varepsilon = .01$). The x-axis shows the area of range queries as the percentage of total area of the universe $[0, R]^2$ (e.g., 4% represents queries with area $= 0.04 \cdot R^2$). Each error is shown as a fraction of stream size n.

in Subsection 3.2. We compared the approximation error of this scheme with a random sample (RS) based approximation. A fixed size RS over a data stream can be generated with the one-pass sampling technique of Vitter [34].

For this experiment, we fixed $n = 10^6$ and $\varepsilon = .01$. We built the ASP tree over n points and then we chose a random sample of size equal to the size of the ASP tree. We chose the query ranges of several different sizes (2% to 20% of the total area of the box $[0, R]^2$). The error in estimated range count is (trueCount − estimatedCount). We computed the average errors for each fixed range area. Figure 4 shows the variation of the average error with range size. The results show that for all three datasets, ASP is an order of magnitude more accurate than RS. We tried the same experiment with $\varepsilon = .1$ and .001 and got similar results. We also note that as the query area increases, the performance of RS degrades, whereas ASP delivers the same error quality.

5.4 Computing Quantiles

Although ASP is a general-purpose representation of multidimensional data, it is interesting to compare its performance on 1-dimensional data to that of problem-specific 1-dimensional algorithms. We used ASP to compute 1-dimensional quantiles, using a stream of integer values randomly chosen from the range $[1, 2^{20}]$. We compared the results with two specialized schemes for quantile computation: Greenwald–Khanna's algorithm [18] (GK) and Manku, Rajagopalan, and Lindsay's algorithm [23] (MRL).

	MRL		ASP		GK	
n	Size	Error ($\times 10^{-4}$)	Size	Error ($\times 10^{-4}$)	Size	Error ($\times 10^{-4}$)
10^5	8334	4.69	2027	5.59	919	8.48
10^6	15155	3.27	2047	4.01	919	8.00
10^7	27475	2.35	2047	6.60	919	7.82

Table 2: Quantile approximation for streams.

We fix $\varepsilon = 10^{-3}$ and maintain the ASP summary for streams of size n=10^5, 10^6 and 10^7. We compute the quantiles of the input at rank $\frac{i}{16}n$ for $i = [1..15]$. The error in computed rank is (trueRank − computedRank). We take the average of these errors for each scheme. Table 2 shows the variation of the size and the average error with n. Each error is shown as a fraction of the stream size n. (The value 4.01 in the second row of ASP means the rank error was $4.01 \times 10^{-4} \times n = 401$). It is clear from the comparison that ASP outperforms the MRL algorithm, and shows comparable performance to the GK

algorithm, even though it is not tuned for the quantiles problem.

6 Conclusion

We have proposed a novel summary for a stream of multidimensional points. This scheme provides a general-purpose summary of the spatial distribution of the points and can answer a wide variety of queries. In particular, ASP can be used to solve spatial analogues of several classical data stream problems, including hot spots and approximate range counting. When specialized to single-dimensional data, adaptive spatial partitioning provides alternative schemes for finding the most frequent items and approximate quantiles, with performance comparable to previous algorithms. Our scheme extends to the sliding window model with a $\log(\varepsilon n)$ factor increase in space.

References

[1] P. K. Agarwal, S. Har-Peled, and K. R. Varadarajan. Approximating extent measures of points. *J. ACM* **51** (2004), pp. 606–635.

[2] N. Alon, Y. Matias, M. Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.* **58** (1999), 137–147.

[3] A. Arasu and G. Manku. Approximate counts and quantiles over sliding windows. In *Proc. 23rd PODS*, 2004, ACM Press, pp. 286–296.

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *21st PODS*, 2002, ACM Press, pp. 1–16.

[5] A. Bagchi, A. Chaudhary, D. Eppstein, and M. T. Goodrich. Deterministic sampling and range counting in geometric data streams. ACM Computing Research Repository, `cs.CG/0307027`.

[6] J. L. Bentley. Multidimensional divide-and-conquer. *Communications of the ACM* **23** (4) (1980) 214–229.

[7] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 2nd edition, 2000.

[8] M. Charikar, L. O'Callaghan, and R. Panigrahy. Better streaming algorithms for clustering problems. In *Proc. 35th STOC*, 2003, pp. 30–39.

[9] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Finding hierarchical heavy hitters in data streams. In *Proc. 29th Conf. VLDB*, 2003.

[10] G. Cormode and S. Muthukrishnan. Radial histograms for spatial streams. Technical report DIMACS TR 2003-11, 2003.

[11] G. Cormode and S. Muthukrishnan. What is hot and what is not: Tracking most frequent items dynamically. *Proc. 22nd PODS*, 2003, 296–306.

[12] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM Journal of Computing* **31** (6) (2002), 1794–1813.

[13] L. Golab, D. DeHaan, E. D. Demaine, A. Lopez-Ortiz, and J. I. Munro. Identifying frequent items in sliding windows over on-line packet streams. In *Proc. Conf. Internet Measurement*, ACM Press, 2003, 173–178.

[14] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Frequency estimation of internet packet streams with limited space. *Proc. 10th ESA*, LNCS 2461, 2002, pp. 348–360.

[15] C. Estan, S. Savage and G. Varghese. Automatically inferring patterns of resource consumption in network traffic. In *Proc. SIGCOMM*, ACM Press, 2003, pp. 137–48.

[16] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *Proc. 24rd Conf. VLDB*, 1998, pp. 299–310.

[17] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. How to summarize the Universe: Dynamic maintenance of quantiles. In *Proc. 28th Conf. on VLDB*, 2002.

[18] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *Proc. 20th SIGMOD*, 2001, pp. 58–66.

[19] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. E. Tarjan, Linear time algorithms for visibility and shortest path problems inside simple polygons. *Algorithmica* **2** (1987), 209–233.

[20] J. Hershberger and S. Suri. Adaptive sampling for geometric problems over data streams. In *Proc. 23rd PODS*, 2004, ACM Press, pp. 252–262.

[21] R. M. Karp, S. Shenker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems* **28** (1) (2003), 51–55.

[22] G. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proc. 28th Conf. VLDB*, 2002, pp. 346–357.

[23] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *Proc. 17th SIGMOD*, 1998, pp. 426–435.

[24] G. Manku, S. Rajagopalan, and B. G. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. *Proc. 18th SIGMOD*, 1999, pp. 251–262.

[25] J. Misra and D. Gries. Finding repeated elements. *Sci. Comput. Programming* **2** (1982), 143–152.

[26] J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. *Theoretical Computer Science* **12** (1980), 315–323.

[27] S. Muthukrishnan. Data streams: Algorithms and applications. Preprint, 2003.

[28] M. H. Overmars. Design of dynamic data structures. Vol. 156 of LNCS, Springer, Berlin, 1987.

[29] C. Schwarz, M. Smid, J. Snoeyink, An optimal algorithm for the on-line closest pair problem, *Algorithmica* **12** (1994), 18–29.

[30] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and beyond: New aggregation techniques for sensor networks. *Proc. of the Second ACM Conference on Embedded Networked Sensor Systems*, 2004.

[31] S. Suri, Cs. D. Tóth, and Y. Zhou, Range counting over multi-dimensional data streams. *Proc. 20th ACM Symp. Comput. Geom.*, ACM Press, 2004, pp. 160- 169.

[32] N. Thaper, S. Guha, P. Indyk, and N. Koudas. Dynamic multidimensional histograms. In *Proc. SIGMOD Conf. on Management of Data*, ACM Press, 2002, 428–439.

[33] V. N. Vapnik and A. Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory Probab. Appl.* **16** (1971), 264–280.

[34] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Software* **11** (1985), 37–57.

[35] W. Wang, J. Yang, and R. Muntz. PK-tree: A spatial index structure for high dimensional point data. In *Proc. 5th Int. Conf. on Foundation of Data Organization (FODO)*, 1998, pp. 281–293.