Range Thresholding on Streams

Miao Qiao
Massey University
New Zealand
m.qiao@massey.ac.nz

Junhao Gan
University of Queensland
Australia
j.gan@uq.edu.au

Yufei Tao
University of Queensland
Australia
taoyf@itee.uq.edu.au

ABSTRACT

This paper studies a type of continuous queries called *range thresholding on streams* (RTS). Imagine the stream as an unbounded sequence of elements each of which is a real value. A query registers an interval, and must be notified as soon as a certain number of incoming elements fall into the interval. The system needs to support multiple queries simultaneously, and aims to minimize the space consumption and computation time.

Currently, all the solutions to this problem entail quadratic time $O(nm)$ to process n stream elements and m queries, which severely limits their applicability to only a small number of queries. We propose the first algorithm that breaks the quadratic barrier, by reducing the computation cost dramatically to $O(n + m)$, subject only to a polylogarithmic factor. The algorithm is general enough to guarantee the same on weighted versions of the queries even in d-dimensional space of any constant d. Its vast advantage over the previous methods in practical environments has been confirmed through extensive experimentation.

CCS Concepts

•Information systems → Stream management;

Keywords

Range Thresholding; Streams; Algorithms

1. INTRODUCTION

Stream processing is an important topic, especially in today's big-data environments where applications with gigantic update volumes abound. *Continuous queries* [1, 2, 8, 17, 21] constitute a major analytical tool on stream data. Unlike queries in a traditional DBMS whose results are determined by the current state of the database, a continuous query is a long-running operation that monitors the incoming traffic in a non-stop manner, and triggers time-critical actions when certain conditions are fulfilled.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2915965

This paper studies a new type of continuous queries that we call *range thresholding on streams* (RTS). For a basic one-dimensional version, consider the stream as an unbounded sequence of elements $e_1, e_2, e_3, ...$, where each element e_i ($i \geq 1$) carries:

- a *value* $v(e_i)$, which is a real number
- a *weight* $w(e_i)$, which is a positive integer.

A query q specifies an interval $R_q = [x, y]$ and an integer $\tau_q \geq 1$. Let $S(q)$ be the set of stream elements e that

- are received after q is registered in the system, and
- have value $v(e)$ falling in R_q.

The query *matures* when the total weight of the elements in $S(q)$ exceeds τ_q for the first time. The system's job is to capture the maturity of q in real time. The challenge is to scale well to a huge number of simultaneous queries, in terms of both space and computation cost.

An RTS query, intuitively, detects when the interval R_q becomes a "hot spot", by having accumulated a total weight of τ_q. It is a crucial building brick in applications that demand rapid responses to such hot spots. As an example, in stock trading, a fund manager pays close attention to the trading volumes in sensitive price ranges. In particular, substantial selling may be an early sign of a significant price downfall. The manager must receive timely alerts about this, for which purpose a typical RTS query q has the form:

"Alert me when $\tau_q = 100,000$ shares of AAPL (Apple Inc.) have been sold in the price range $R_q = [100, 105]$ from now."

Relevant to the query is the stream where each element e is an AAPL trading transaction with $v(e)$ equal to the selling price, and $w(e)$ equal to the number of shares sold. The maturity of such a query is a key factor in various financial models, whose outcome plays a vital role in trading decisions. The system must support a large number of queries of this sort, each with its own R_q and τ_q.

RTS may look deceptively easy at first glance: whenever a new element e arrives, check whether $v(e)$ is in R_q; if so, increase the weight of R_q by $w(e)$—clearly, a constant time operation overall. This method, unfortunately, scales poorly to the number m of queries running at the same time: now it costs $O(m)$ time to process e. The total cost of processing n stream elements thus becomes a prohibitive quadratic term $O(nm)$. This implies that the method is computationally intractable when m is large. Surprisingly, in spite of the fundamental nature of RTS and the obvious defects of "conventional solutions" like the above, currently no progress has been made to break the quadratic barrier.

Even less is understood when the dimensionality d departs from 1, in which case $v(e)$ and R_q become respectively a point and a rectangle in d-dimensional space, as in the following query q:

"Alert me when $\tau_q = 100,000$ shares of AAPL have been sold by transactions e satisfying

- *the selling price of e is in* $[100, 105]$*, and*
- *when e takes place, the NASDAQ index is at 4,600 or lower."*

Here, $v(e)$ is a point (price, NASDAQ), whereas R_q is the rectangle $[100, 105] \times (-\infty, 4600]$ in the same two-dimensional space. It is straightforward to design an $O(nm)$ time algorithm, but how to escape the quadratic trap is even more elusive.

Our Contributions. We present the first algorithm that circumvents the quadratic barrier for any constant dimensionality d. Our algorithms process m queries and n stream elements in a total of $\tilde{O}(n+m)$ time—where notation $\tilde{O}(.)$ hides a polylogarithmic factor—namely, $\tilde{O}(1)$ time per query and element amortized. Apparently, $\Omega(n+m)$ time is compulsory even just to see each query and element once. Therefore, our algorithm is asymptotically optimal up to only a polylogarithmic factor.

Perhaps even more interesting are the techniques we have devised to bring about the major improvement. The most crucial discovery is an observation on the connection between RTS and another surprisingly remote problem called *distributed tracking* [9] (a detailed account of which will appear in Section 3.1). The observation then leads to the development of a novel algorithmic paradigm which, we believe, serves as a powerful weapon for implementing real-time triggers such as RTS.

This paper also features an extensive experimental evaluation of the proposed techniques. The results are fully aligned with our theoretical analysis: our algorithm is faster than conventional approaches by a factor up to orders of magnitude.

Paper Organization. Section 2 formally defines the RTS problem. Section 3 reviews the previous work related to ours. Sections 4-7 describe our RTS algorithms, and prove their theoretical guarantees. Section 8 empirically compares the proposed solutions to alternative methods. Finally, Section 9 concludes the paper with a summary of our findings.

2. PROBLEM FORMULATION

We now encapsulate all the definitions in Section 1 into a coherent framework. Given a constant integer $d \geq 1$, we consider the *data space* $\mathbb{R}^d$, i.e., the d-dimensional Euclidean space where each dimension has a real-valued domain $\mathbb{R}$.

The *data stream* is an unbounded sequence of elements, where the i-th ($i \geq 1$) element is denoted as e_i, and is said to arrive at *time* i. Each element e carries two fields:

- A point $v(e)$—called the *value* of e—in the data space;
- A *weight* $w(e)$ which is a positive integer.

An RTS query q specifies a d-dimensional axis-parallel rectangle R_q and an integer *threshold* $\tau_q \geq 1$. If the query is issued after receiving e_j (for some $j \geq 1$) and t is an integer larger than j, we use $S(q,t)$ to represent the set of elements among e_{j+1}, e_{j+2}, ..., e_t that are covered by R_q, i.e.,

$$S(q,t) = \{e_i \mid j < i \leq t \text{ and } e_i \in R_q\}.$$

Define:

$$W(q,t) \quad = \sum_{e \in S(q,t)} w(e) \tag{1}$$

which sums up the weights of all the elements in $S(q,t)$. Since every $w(e)$ is positive, $W(q,t)$ monotonically increases with time. The *maturity time* of q is the smallest j' making the following hold

$$W(q,j') \geq \tau_q$$

at which moment the query *matures*.

Let n be total number of stream elements so far, i.e., the *length* of the stream at this point. For convenience, we abbreviate $S(q,n)$ as $S(q)$, and $W(q,n)$ as $W(q)$.

The system allows two operations:

- REGISTER(q): Accept a query at the current moment (after the arrival of e_n).
- TERMINATE(q): Stop and eliminate a query.

A query, which has been registered but not terminated, is *alive*. We denote by Q the set of alive queries. For every query $q \in Q$, the system *must* report the maturity of q at its maturity time. Such a query is then automatically removed from the system with a TERMINATE operation.

Denote by m the total number of queries ever registered in history. The system's efficiency goal is to *entail total computation cost of* $\tilde{O}(n+m)$. In addition, we require that it should consume $\tilde{O}(m_{alive})$ space at all times, where m_{alive} is the number of queries alive currently. This essentially implies that one cannot hope to store too many stream elements (if any at all).

This completes the definition of the RTS problem. A special instance—which we refer to as *counting RTS*—arises when $w(e) = 1$ for all the stream elements e. Later, we will discuss first the counting version, which allows us to elaborate on our central ideas, before including the extra details to solve the general RTS problem.

3. RELATED WORK

Section 3.1 discusses how RTS can be tackled using known techniques. Section 3.2 presents an introduction to the *distributed tracking* problem. Finally, Section 3.3 surveys other research that is relevant, but to a less extent.

3.1 Existing RTS Solutions

Section 1 has already mentioned a *baseline* approach for solving the RTS problem, which maintains the precise value of $W(q)$ for all alive queries q. Given an incoming stream element e, it simply checks whether $v(e) \in R_q$, and if so, increases $W(q)$ accordingly. This approach demands the minimum amount of space $O(m_{alive})$, but incurs $O(m_{alive})$ time processing e. It is easy to see that in the worst case its overall computation time can be as high as $O(nm)$.

Query indexing has become a standard method to tackle continuous queries. The rationale is to create an index structure on the *queries* (unlike a conventional database where indexes are on *data*). Each arriving stream element trickles down the index, and updates the relevant information therein. This motivates a better RTS approach that maintains a structure on the set Q of alive queries to support the *stabbing operation*:

> Given a point v in the data space $\mathbb{R}^d$, report the set of queries $q \in Q$ such that $v \in q$. These queries are said to be *stabbed* by v.

Such operations have been very well studied (see [20] for a full summary of the past results). In one-dimensional space, the *interval tree* [12] consumes $O(|Q|) = O(m_{alive})$ space, supports an insertion/deletion in $\tilde{O}(1)$ time, and performs a stabbing operation in $\tilde{O}(1+k)$ time, where k is the number of queries reported. Using standard layering techniques (see Chapter 5.3 of [12]), the interval tree can be extended to d-dimensional space, by paying a logarithmic factor in the space, update, and query overhead. For constant d, the resulting structure uses $\tilde{O}(m_{alive})$ space, supports an update in $\tilde{O}(1)$ time, and allows a stabbing operation in $\tilde{O}(1+k)$ time. For practical data, the *R-tree* [5, 16] is a dynamic structure that may exhibit satisfactory efficiency for the stabbing operation, although no attractive worst-case guarantees exist on updates and queries.

Given such an index, the *stabbing approach* launches, for each stream element e, a stabbing operation to find the queries stabbed by $v(e)$. Every such query q then has its $W(q)$ increased by $w(e)$. The stabbing approach is still captivated by the quadratic trap because the value of k—the number of queries stabbed by an incoming element—can be as large as m in the worst case, and thereby, necessitating $O(nm)$ time overall. However, better efficiency is expected in practice because, with a more careful analysis, one can easily show that the cost of the approach is bounded by $\tilde{O}(n) + O(m \cdot \tau_{max})$, where τ_{max} is the largest threshold of all queries. When viewed this way, the quadratic trap takes up a different form, i.e., with respect to m and τ_{max}.

3.2 Distributed Tracking

This problem is defined in a distributed environment with $h+1$ sites: one of them is the *coordinator* q, while the rest are *participants* $s_1, s_2, ..., s_h$. Communication exists between only the coordinator and the participants, whereas the participants do not talk to each other.

Each participant s_i ($1 \le i \le h$) has an integer counter c_i. Initially, all the counters $c_1, c_2, ..., c_h$ are 0. At every timestamp, at most one of those counters gets increased by 1—in other words, it is possible that no counter is incremented, but never more than one. The coordinator's job is to report *maturity* when the following condition holds

$$\sum_{i=1}^{h} c_i = \tau$$

where $\tau > 0$ is an integer. To achieve the purpose, an algorithm instructs the sites to send messages strategically. The goal is to minimize the messaging cost, measured in the number of words transmitted.

A straightforward solution to this problem is to have a participant inform the coordinator whenever its counter increases. The total communication cost is apparently τ bits, which is expensive if τ is large. The problem admits another algorithm [9] that requires the communication of $O(h \log \tau)$ words. Note that this is far less than τ bits when τ is (realistically) much larger than h. Next, we describe the algorithm in full.

First, if $\tau \le 6h$, then solve the problem using the straightforward solution by sending $O(\tau) = O(h)$ words.

Consider now $\tau > 6h$. The coordinator q sends to every participant s_i ($1 \le i \le h$) the value which we call the *slack*:

$$\lambda = \lfloor \tau/(2h) \rfloor. \qquad (2)$$

This requires h messages. Then, s_i intermittently sends a one-bit *signal* to q according to the following rule:

> Let $\bar{c}_i$ be the counter value at the time of the last signal to q (0, if no signal yet). Send a signal, as soon as $c_i - \bar{c}_i = \lambda$.

The coordinator keeps track of the number of signals received. When the number reaches h, it collects the precise counters from all the participants using h messages, and calculates:

$$\tau' = \tau - \sum_{i=1}^{h} c_i.$$

This finishes a *round*.

Notice that, at this moment, we are facing the same problem but with a reduced $\tau = \tau'$. The algorithm recursively solves it as above. Eventually, the value of τ drops to at most $6h$, in which case the problem is settled with a final round using the straightforward approach.

Each round clearly necessitates $O(h)$ messages. It is not hard to verify from $\tau > 6h$ that $\tau' \le 2\tau/3$, implying that the total number of rounds is $O(\log \tau)$. This explains why the total messaging cost is $O(h \log \tau)$ words.

3.3 Other Relevant Research

This subsection reviews several topics related to RTS, paying an effort to provide a chronological view on their development.

Triggers, a form of which RTS can be though of as, are one of the oldest concepts in database systems. As taught in standard textbooks, a conventional trigger typically serves a "passive" role, which is to ensure certain integrity constraints on the underlying relations. In the 1990's, the community started to introduce novel, more sophisticated, forms of triggers that allow a database to activate itself in response to a great variety of events. The functionalities of those "advanced" triggers eventually escalate to a level such that people decided to call them *event-condition-action* rules. Efficient implementation of such rules in DBMS has been extensively studied in the literature of *active databases*; see [11, 19] for a nice introduction.

Early research assumed datasets with low insertion/deletion frequency. Since the emergence of update-intensive applications at the beginning of the century, stream processing has been a major subfield in the database area. Continuous queries—a concept that, interestingly, had its first appearance in non-stream days [21]—are a main focus in the research of modern systems for managing stream data (e.g., [1, 2, 8, 17]). Our work also belongs to this line of research.

Triggers were revitalized on streams. This was reflected not only in the community's endeavor to *implement* triggers to meet the new efficiency requirements [7], but also in the creative *use* of triggers to build innovative application platforms, perhaps most notably, *publish/subscribe systems*; see some representative works [8, 14, 15, 18, 23] and the references therein. Such a system is built on a stream of elements (e.g., tweets, news articles, product orders, etc), only some of which may be interesting to a user. Instead of "publishing" everything, the system allows the user to "subscribe" to particular information. A subscription is essentially a trigger, specifying when the system should *push* information to the user, and if so, what information to push.

RTS, if not treated entirely as a continuous *query*, may also be counted as a stream trigger, or a form of content subscription. In the sense of the former, the user issuing an RTS query should get an alert at the query's maturity time. In the sense of the latter, on the other hand, the user implicitly subscribes to a form of information *aggregated* from the elements flowing through the system. Regardless of the perspective, however, RTS is unique in its own, especially given the fact that there does not exist an efficient algorithm—not counting the ones in Section 3.1 with the quadratic time complexity.

Finally, we note that active databases have also be rejuvenated on streams, yielding another line of research under the theme of *complex event processing*; see [10, 13, 22] for entry points of reading into this subfield. Our work complements that subfield by adding RTS as another "efficiently-supportable" atomic event type.

4. THE FIRST ALGORITHM

We will incrementally unfold our ultimate RTS algorithm. Let us start by constraining the problem in two ways:

- Counting RTS with $d = 1$, that is, the value of each stream element is one-dimensional, and its weight is 1.
- All the queries are registered at the beginning, namely, before receiving the first element. No query is accepted afterwards. Those queries remain alive until maturity. This is called the *one-time registration constraint*.

The goal of this section is to settle the above restricted problem using $\tilde{O}(m_{alive})$ space at any moment, and $\tilde{O}(n+m)$ time overall. The imposed constraints permit us to concentrate on explaining the most crucial ideas. Both constraints will be eventually removed in later sections; and our final algorithm still achieves the same space and query complexities.

To simplify our technical exposition, we will treat each query interval R_q as $[x, y)$, namely, open at the right end. This assumption does not lose generality because, as a standard technique, a closed interval $[x, y]$ can be regarded as $[x, y + \epsilon)$ for an infinitesimal $\epsilon > 0$.

The Endpoint Tree. Define Q^* as the set of registered queries. We create a binary search tree $\mathcal{T}$ on the endpoints of the queries' intervals; see Figure 1 for an example with 8 queries. The height of $\mathcal{T}$ is $O(\log m)$.

Every node u in $\mathcal{T}$ is naturally associated with a *jurisdiction interval* $I(u)$ as follows. Consider first that u is a leaf. Suppose that u stores an endpoint x, and u'—the leaf succeeding u—stores x' ($= \infty$ if u' does not exist); then, $I(u) = [x, x')$. Consider now u as an internal node with child nodes u_1, u_2; then $I(u) = I(u_1) \cup I(u_2)$. As an example, in Figure 1, the jurisdiction interval of leaf 8 is $[8, 9)$, that of leaf 16 is $[16, \infty)$, and that of u_1 is $[5, 9)$.

Let e be a stream element such that $v(e)$ is at least the leftmost endpoint in $\mathcal{T}$ (otherwise, e obviously can be safely ignored). At each level of $\mathcal{T}$, e is *covered* by precisely one node u: the one with $I(u)$ containing $v(e)$. These nodes can be easily found in $O(\log m)$ time by descending a single root-to-leaf path.

Every node u in $\mathcal{T}$ stores a counter $c(u)$, which equals the number of stream elements covered by u. The counters are initially set to 0, and maintained along with the arrival of each element e: if $I(u)$ covers $v(e)$, increase $c(u)$ by 1. We then discard e forever—our structure does not store any elements. It is thus clear that the maintenance takes $O(\log m)$ time per element.

We refer to $\mathcal{T}$ as the *endpoint tree*. It is rudimentary to construct it in $O(m \log m)$ time.

"Distributed" Tracking. We are ready to unveil how to leverage the distributed tracking (DT) algorithm in Section 3.2 for RTS.

Focus, for the time being, on an arbitrary query $q \in Q^*$. Its interval $R_q = [x, y)$ can be partitioned by the jurisdiction intervals of nodes that constitute the *canonical node set* U_q. This is the minimum set of nodes with *disjoint* jurisdiction intervals whose union equals R_q. U_q contains at most 2 nodes per level of $\mathcal{T}$, implying that $U_q = O(\log m)$. For instance, consider query q_5 in Figure 1: $U_{q_5} = \{u_1, u_2, u_3\}$. Given x, y, it is a standard exercise[1] to identify U_q in $O(\log m)$ time.

Next, we define a conceptual DT problem instance for q. Set $h_q = |U_q|$. There are h_q "participants", each of which is a node $u \in U_q$. The counter of "participant" u is simply $c(u)$. The query q itself is the "coordinator", whose mission is to capture maturity when the following condition holds:

$$\sum_{u \in U_q} c(u) \quad = \quad \tau_q.$$

Note that, by definition of U_q, the left hand side of the equation is precisely $W(q)$.

It must be emphasized that there is nothing "distributed" in this conversion. All the processing happens on a single machine in the context of RTS. The "distributed" viewpoint is sheerly to allow the reader to borrow the knowledge from Section 3.2 to see what is actually happening here.

We then "run" the algorithm in Section 3.2 to solve this DT instance, by *simulating* all of its steps. Whenever $c(u)$ increases by 1, we reflect so by thinking that the counter of the "participant" u has gone up by 1. Whenever the algorithm instructs a "participant" u to send a message (of no more than one word) to the "coordinator" q or the other way around, we spend $O(1)$ time to do so explicitly, so that both u and q can maintain the same information as demanded by the algorithm.

A piece of detail deserves a more careful look. In the DT algorithm, every node $u \in U_q$ keeps—as described in Section 3.2—the counter value $\bar{c}_q(u)$ at the time of the last

[1]Let z_1 and z_2 be the leaves storing x and y, respectively. Identify the lowest common ancestor u of z_1, z_2. Then, invoke the following procedure ADD(u). In general, the procedure first checks whether $I(u) \subseteq [x, y)$. If so, it adds u to U_q and finishes. Otherwise, it checks whether $I(u)$ is disjoint with $[x, y)$. If so, it adds nothing and finishes. If not, it recursively invokes ADD(u_1) and ADD(u_2), where u_1 and u_2 are the two child nodes of u.

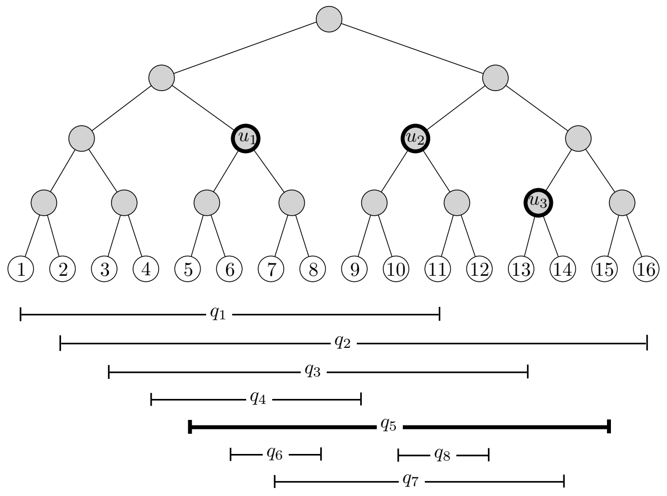


Figure 1: One-dimensional endpoint tree

signal to q. The next signal from u takes place when

$$c(u) - \bar{c}_q(u) \quad = \quad \lambda_q \tag{3}$$

where λ_q is the *slack* value given in (2). Node u needs to inspect the above condition *only* when its counter $c(u)$ changes. Hence, every incoming element triggers such slack inspection only at $O(\log m)$ nodes.

To analyze the computation cost, recall from earlier discussion that the time of maintaining the node counters is $O(n \log m)$. The same cost applies to inspecting the slack condition in (3). The rest of the DT algorithm is simulated with $O(1)$ time per "message". We already know from Section 3.2 that the algorithm needs $O(h_q \log \tau_q)$ messages to solve the DT instance of q. Therefore, in the RTS context, such *messaging cost* adds up to

$$O(h_q \log \tau_q) \quad = \quad O(\log m \cdot \log \tau_q) \tag{4}$$

in total.

Putting Together All Queries with Heaps. Every query $q \in Q^*$ defines a DT instance as above, which is solved by an instance of the algorithm of Section 2. In other words, conceptually, there are m algorithm instances in progress simultaneously. Our task is to implement such "concurrent" execution efficiently on our *sequential* CPU.

Two immediate observations are helpful. First, the maintenance of node counters (i.e., $c(u)$ of every node u in $\mathcal{T}$) is *common* to all instances. Therefore, such maintenance incurs only $O(n \log m)$ time in total for m queries.

Second, the messaging cost of each query q is specific to q itself, and is given by (4). Hence, the total messaging cost of all queries is:

$$\sum_{q \in Q^*} O(\log m \cdot \log \tau_q) \quad = \quad O(m \log m \cdot \log \tau_{max})$$

where τ_{max} the largest τ_q of all $q \in Q^*$.

It remains to discuss the cost of inspecting the slack condition (3). Concentrate on an arbitrary node u. Let $Q(u)$ be the set of queries that have u in their canonical node sets. For example, consider u_2 in Figure 1: $Q(u_2) = \{q_2, q_3, q_5, q_7\}$.

Carrying out the one-query strategy naively, when $c(u)$ changes, we would inspect the condition for all the queries in $Q(u)$. However, this, taking $O(|Q(u)|)$ time, is overly expensive, and will blow up the overall cost essentially to quadratic again. We overcome this issue by inspecting only *one* slack condition, instead of $|Q(u)|$, as explained next.

For each query $q \in Q(u)$, define

$$\sigma_q(u) \quad = \quad \lambda_q + \bar{c}_q(u). \tag{5}$$

Phrased differently, $\sigma_q(u)$ indicates the *future* value of $c_q(u)$ when the next signal from u to q should occur. For two queries $q_1, q_2 \in Q(u)$ such that $\sigma_{q_1}(u) < \sigma_{q_2}(u)$, $\sigma_{q_1}(u) > c(u)$ always implies $\sigma_{q_2}(u) > c(u)$. Namely, if a signal to q_1 is not necessary yet, neither is it to q_2.

Motivated by this, we maintain the $\sigma_q(u)$ of all the queries $q \in Q(u)$ in a min-heap $\mathcal{H}(u)$. Whenever $c(u)$ changes, we perform these steps:

1. Find the minimum $\sigma_q(u)$ in $\mathcal{H}(u)$ in $O(1)$ time.
2. If $c(u) < \sigma_q(u)$, then we are done.
3. Otherwise, remove $\sigma_q(u)$ from $\mathcal{H}(u)$, and instruct u to send a signal to q (as in the DT algorithm). This takes $O(\log |\mathcal{H}(u)|) = O(\log m)$ time. Then, repeat from Line 1.

It is evident that the above steps run in

$$O(1 + x \cdot \log m) \tag{6}$$

time, where x is the number of signals issued at Line 3.

We now bound the total cost of the 3 steps during the entire algorithm, including all nodes and queries. The $O(1)$ term in (6) occurs $O(n \log m)$ times, because each stream element can increase the counters of $O(\log m)$ nodes. To bound the sum of the term $x \log m$, we point out that the total number of signals throughout the algorithm is

$$\sum_{q \in Q^*} O(h_q \log \tau_q) \quad = \quad O(m \log m \cdot \log \tau_{max}). \tag{7}$$

Therefore, all the $x \log m$ terms add up to $O(m \log^2 m \cdot \log \tau_{max})$.

To complete the time analysis, we must note that $\mathcal{H}(u)$ demands additional cost for the insertions and deletions therein. Specifically, whenever the λ_q of a query q changes, so does $\sigma_q(u)$ for each $u \in U_q$; see (5). This incurs $O(\log m)$ time to remove the old $\sigma_q(u)$ from $\mathcal{H}(u)$ and to add back the new one. The number of λ_q changes equals the number of rounds in the DT algorithm for q, which as explained in Section 3.2 is $O(\log \tau_q)$. Hence, the total number of heap insertions and deletions (over all queries, all nodes) is also given by (7). Their cost is therefore $O(m \log^2 m \cdot \log \tau_{max})$.

Finally, as each query adds an entry in the heaps of $O(\log m)$ nodes, the total space consumption is bounded by $O(m \log m)$.

Handling Maturity. When a query q matures, we simply remove its entry from the $\mathcal{H}(u)$ of all nodes $u \in U_q$, in $O(h_q \log m) = O(\log^2 m)$ time. This adds $O(m \log^2 m)$ time to the overall computation cost.

Recall that we aim to use $\tilde{O}(m_{alive})$ space at all times. Our structure, as mentioned earlier, occupies $\tilde{O}(m)$ space, which can break the promise when $m_{alive} \ll m$, namely, after many queries have matured. Next, we explain how to deal with the issue with *global rebuilding*.

When m_{alive} has decreased to $m/2$, we destroy the whole structure, and re-start the entire algorithm from *scratch* with respect to the remaining set Q of alive queries which, importantly, have their thresholds adjusted. Specifically, for each query $q \in Q$, we obtain $W(q)$ (the actual number of elements that have already arrived and fallen into R_q, as defined in Section 2). The value of $W(q)$ equals the sum of the counters of all the nodes in U_q, and thus can be obtained in $O(\log m)$ time. From *now* on, q matures after another $\tau_q - W(q)$ subsequent elements get into R_q. We thus reset its threshold to $\tau_q - W(q)$. After all the threshold resetting is complete, we rebuild $\mathcal{T}$ in $O(m_{alive} \log m_{alive})$ time, after which the algorithm runs as if the stream had just started.

One global rebuilding takes $O(m \log m)$ time in total. This can be amortized on the $m/2$ queries that have matured, so that each of those query bears $O(\log m)$ time. As each query bears such cost only once, the total rebuilding cost is $O(m \log m)$ for the entire algorithm.

If Q^* is the set of queries alive at the moment of the most recent reconstruction, the above approach ensures that $m_{alive} \geq |Q^*|/2$. Therefore, our space consumption now becomes $O(|Q^*| \log |Q^*|) = O(m_{alive} \log m_{alive})$, as desired.

Remark. We now have solved the simplified RTS problem defined at the beginning of this section. Our algorithm guarantees $O(m_{alive} \log m_{alive}) = \tilde{O}(m_{alive})$ space at any moment, and performs $O(n \log m + m \log^2 m \cdot \log \tau_{max}) = \tilde{O}(n + m)$ time in total.

5. DYNAMIC QUERIES

This section will remove the "one-time registration" constraint, where queries are *static* in the sense that all of them need to be registered at the beginning of the stream. Next, we extend our algorithm of the previous section to the *dynamic* scenario, allowing the REGISTER and TERMINATE operations defined in Section 2 at any time.

Termination. The data structure behind our algorithm—namely the endpoint tree—easily supports a TERMINATE(q) operation, in the same manner as removing a matured query. Specifically, it suffices to delete the entry of q from the $\mathcal{H}(u)$ of all nodes $u \in U_q$. This takes $O(\log^2 m)$ time as explained before. Note that the underlying binary search tree $\mathcal{T}$ does not change after the termination (because $\mathcal{H}(u)$ is a secondary min-heap associated with u). After $m/2$ queries have disappeared (either matured or deleted), apply global rebuilding as described in Section 4 to maintain the space consumption $\tilde{O}(m_{alive})$.

Registration. Recall that $\mathcal{T}$ is built on the endpoints of the query intervals. To terminate a query, we can afford to retain the endpoints of the query's interval in $\mathcal{T}$, because this does not affect the algorithm's correctness (some redundancy is left in the structure, but the extra space is affordable, as proved earlier). To insert a new query with REGISTER(q), however, we must add the endpoints of R_q into $\mathcal{T}$; otherwise, the node counters in $\mathcal{T}$ do not allow us to derive $W(q)$ precisely, making it impossible to detect its maturity.

Inserting the endpoints of R_q may trigger *rebalancing operations* of $\mathcal{T}$ (i.e., splits or rotations, depending on the binary tree implementation). Accordingly, the secondary min-heaps will need to be reorganized, because rebalancing may disrupt the canonical node sets of many queries. For inquisitive readers, we mention that in theory this can be dealt with using *weight balancing techniques* (see, e.g., [4]), but the algorithm becomes too complicated to implement in practice.

We circumvent this problem by resorting to another, much more practical, algorithmic technique called the *logarithmic method* [3, 6]. It is a generic framework that allows one to convert a *semi*-dynamic structure supporting only deletions to a *fully*-dynamic structure that is able to support both deletions and insertions. In what follows, we explain how to instantiate the framework in our RTS context.

Our indexing scheme now consists of g endpoint trees $\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_g$ satisfying all the properties below:

P1 $g = O(\log m)$.

P2 Every alive query is managed by one and exactly one tree.

P3 Let $m_{alive}(i)$ be the number of alive queries in $\mathcal{T}_i$ ($1 \leq i \leq g$). It must hold that $m_{alive}(i) \leq 2^{i-1}$. Note that $\mathcal{T}_i$ is allowed to be an empty tree (i.e., merely a placeholder) with $m_{alive}(i) = 0$.

At the very beginning, $g = 1$, and $\mathcal{T}_1$ is an empty tree. Consider, in general, that the system receives a REGISTER(q) operation. We handle it with these steps:

1. Identify the smallest $j \geq 1$ satisfying

$$\sum_{i=1}^{j} m_{alive}(i) \quad < \quad 2^{j-1}. \tag{8}$$

   If j does not exist, then set $j = g + 1$ (as we will see, in this case a new endpoint tree will be created).

2. Collect all the alive queries in $\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_j$, together with the new query q, into a set Q^*. For each query $q' \in Q^*$, obtain $W(q')$ from the endpoint tree where it belongs. Reduce the threshold $\tau_{q'}$ of q' by $W(q')$.

3. Discard all of $\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_j$. Construct $\mathcal{T}_j$ on Q^* from scratch. From now on, $\mathcal{T}_j$ is responsible for the queries of Q^* (with thresholds *modified*), treating them as if they were new with respect to the subsequent stream.

   Note that (8) ensures that $\mathcal{T}_j$ obeys Property ***P3***. Furthermore, $\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_{j-1}$ are now all empty.

Given an incoming element e, we now use it to increase the node counters of all g trees. As the counter increasing for each tree takes $O(\log m)$ time, in total the processing time of e is bounded by $O(g \log m) = O(\log^2 m)$. The time is thus $O(n \log^2 m)$ for all the n stream elements.

Next, we analyze the cost of rebuilding the endpoint trees. Our algorithm ensures that if a query moves from $\mathcal{T}_i$ to $\mathcal{T}_{i'}$, it is always due to Line 3, which must have destroyed T_i and rebuilt $\mathcal{T}_{i'}$. This implies that $i' \geq i$ always holds, namely, a query *never* moves from a higher "ranked" endpoint tree to a lower one.

Line 3, whose cost dominates the other steps, builds $\mathcal{T}_j$ in $O(|Q^*| \cdot \log |Q^*|) = O(2^j \log m)$ time. On the other hand, the definition of j (c.f. (8)) implies

$$\sum_{i=1}^{j-1} m_{alive}(i) \quad \geq \quad 2^{j-2} \tag{9}$$

otherwise, j would have been smaller by at least 1. In other words, at least 2^{j-2} queries have moved from a lower ranked

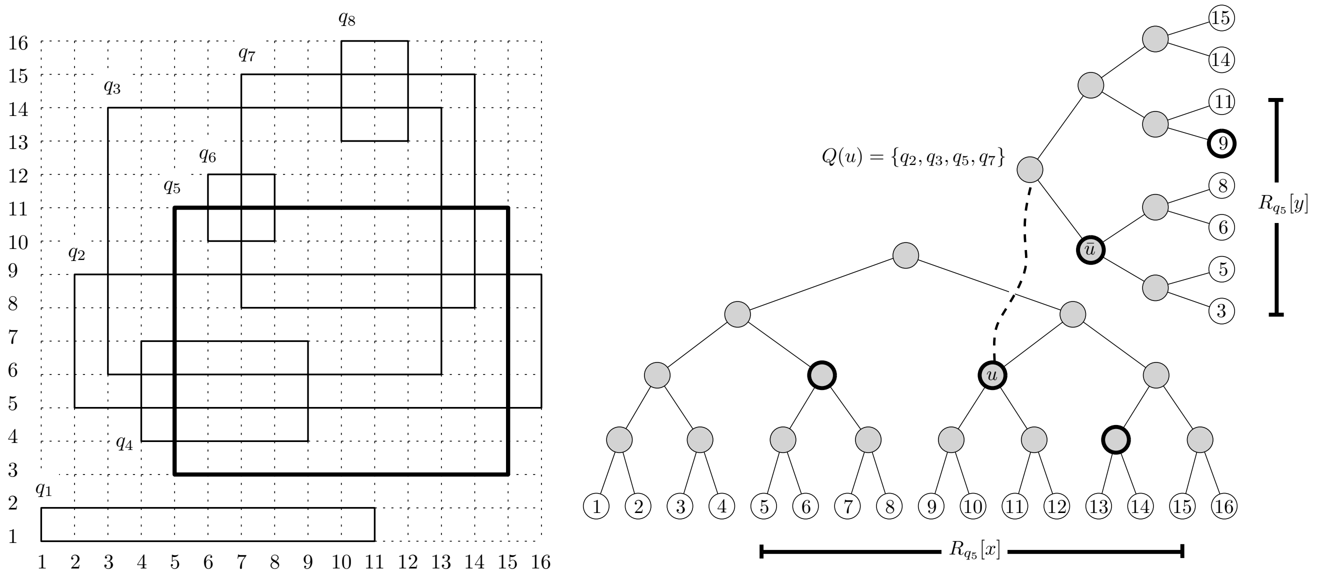


Figure 2: Two-dimensional endpoint tree

endpoint tree to $\mathcal{T}_j$. We can thus charge the $O(2^j \log m)$ construction time of $\mathcal{T}_j$ on those queries, each of which bears only $O(\log m)$. Property ***P*1** tells us that a query can be charged this way only $O(\log m)$ times, i.e., $O(\log^2 m)$ time in total. Therefore, the overall cost of rebuilding the endpoint trees is $O(m \log^2 m)$.

Why can our algorithm ensure ***P*1**? Consider the moment when $\mathcal{T}_g$ has just been constructed at Line 3. (9) shows that $\mathcal{T}_g$ must contain more than 2^{g-2} queries. It thus follows that $m \geq 2^{g-2}$, and hence, $g = O(\log m)$.

None of the other parts of the time analysis in Section 5 is affected. In other words, our algorithm entails $O(n \log^2 m + m \log^2 m \cdot \log \tau_{max})$ time to process a stream of length n and m *dynamic* queries. As each endpoint tree uses space that is proportional to the number of alive queries therein (up to a logarithmic factor), Property ***P*2** ensures that the total space consumption is $\tilde{O}(m_{alive})$ at all times.

6. MULTIDIMENSIONAL QUERIES

We will describe how to solve, still, counting RTS, but in d-dimensional space $\mathbb{R}^d$ with a constant d. This, in fact, has become fairly easy under the algorithmic framework laid out in the previous sections. It should have become clear that there are two central ideas:

- Partition the search region R_q (i.e., an interval for $d = 1$) into $\tilde{O}(1)$ components, for each of which the number of elements within is kept precisely at a node.
- Many queries share the same components such that the total number of distinct components is $\tilde{O}(m_{alive})$.

Plugging almost any such partitioning scheme into our framework will yield an efficient algorithm. For $d = 1$, our scheme—based on canonical node sets—assumes a binary search tree. A natural extension to general d is to use ideas from the *range tree* (see Chapter 5.3 of [12]).

The subsequent discussion explains this for $d = 2$ because the generalization to higher d will then become straightforward. We will first work with the one-time registration constraint, namely, all the m queries are registered at the beginning of the stream. The constraint can be removed by the same technique in Section 5. Once again, for our technical exposition, we regard each R_q in the form of $[x_1, x_2) \times [y_1, y_2)$, with the "infinitesimal trick" explained in Section 4.

Two-Dimensional Endpoint Tree. Let Q^* be the set of all queries. Remember that, for each $q \in Q^*$, R_q is now a two-dimensional rectangle. Denote by $R_q[x]$ the projection of R_q onto the x-dimension, and by $R_q[y]$ the projection onto the y-dimension.

We create a binary search tree $\mathcal{T}$ on the endpoints of $R_q[x]$ of all $q \in Q^*$. As before, each query defines an *x-canonical node set*, which is the minimum set of nodes in $\mathcal{T}$ with disjoint jurisdiction intervals whose union is $R_q[x]$.

Focus now on a specific node u in $\mathcal{T}$. Denote by $Q(u)$ the set of queries that have u in their x-canonical node sets. We associate u with a secondary binary search tree $\mathfrak{T}(u)$, which indexes the endpoints of $R_q[y]$ of all $q \in Q(u)$.

Figure 2 illustrates this using an example with 8 queries—for the reader's convenience, the x-projections of the rectangles are exactly the intervals in our one-dimensional example of Figure 1. For the node u as indicated, $Q(u) = \{q_2, q_3, q_5, q_7\}$. Therefore, $\mathfrak{T}(u)$ is created on the endpoints of the y-projections of the four queries: $[5, 9), [6, 14), [3, 11)$, and $[8, 15)$.

$\mathcal{T}$ and all the the secondary trees $\mathfrak{T}(u)$ constitute our two-dimensional *endpoint tree*. It is rudimentary to construct the structure in $O(m \log m)$ time.

Query Decomposition. It is important to note the *geometric region* in $\mathbb{R}^2$ corresponding to every node u in $\mathcal{T}$, and to every node $\bar{u}$ in a secondary tree $\mathfrak{T}(u)$. Specifically, the jurisdiction interval $I(u)$—which is on the x-dimension—defines the vertical slab $I(u) \times (-\infty, \infty)$. The jurisdiction interval $I(\bar{u})$—which is on the y-dimension—defines the rectangle $I(u) \times I(\bar{u})$. For instance, consider node u and $\bar{u}$ in Figure 2: u corresponds to the slab $[9, 13) \times (-\infty, \infty)$, and $\bar{u}$ to $[9, 13) \times [3, 9)$.

We are ready to explain how the R_q of a query $q \in Q^*$

is partitioned by the above geometric regions. Let u be a node in the x-canonical node set of q. Then, q and u together define $U_q(u)$, which is the minimum set of nodes in $\mathcal{T}(u)$ with disjoint jurisdiction intervals whose union is $R_q[y]$. Consider, for instance, query q_5 and node u of $\mathcal{T}$ in Figure 2: $U_{q_5}(u) = \{\bar{u}, \text{leaf } 9\}$. The *y-canonical node set* of q—denoted as U_q—unions the $U_q(u)$ of all such nodes u.

The size of U_q is $O(\log^2 m)$, because $|U_q(u)| = O(\log m)$ while there are $O(\log m)$ such u. Every node $\bar{u} \in U_q$ is in some secondary tree; and the geometric regions of all the $\bar{u}$ form a disjoint partition of R_q.

Algorithm. Every node $\bar{u}$ in a secondary tree keeps a counter equal to the number of stream elements e such that $v(e)$ falls into the geometric region of $\bar{u}$. These counters can be easily maintained in $O(\log^2 m)$ time per element (descending one root-to-leaf path Π of $\mathcal{T}$, and then one root-to-leaf path of $\mathcal{T}(u)$ for each node $u \in \Pi$). The counter maintenance takes $O(n \log^2 m)$ time in total for n elements.

The rest of the algorithm directly follows that of Section 4. In particular, every query still defines a distributed tracking instance involving the h_q nodes in U_q, where $h_q = |U_q|$. The total number of heap insertions and deletions is still given by the left hand side of (7). As h_q here becomes $O(\log^2 m)$, the total cost of all the heap updates is bounded by $O(m \log^3 m \cdot \log \tau_{max})$.

Eliminating the one-time registration constraint with the logarithmic method (Section 5) increases the counter maintenance overhead by a logarithmic factor. Therefore, our two-dimensional algorithm entails the overall computation time of $O(n \log^3 m + m \log^3 m \cdot \log \tau_{max})$. The space consumption is $O(m_{alive} \log^2 m_{alive})$: essentially the total size of U_q of all the alive queries q.

Remark. Extending our algorithm to general d-dimensional space is at the standard expense of an extra logarithmic factor in both the running time and space per dimension: $O(n \log^{d+1} m + m \log^{d+1} m \cdot \log \tau_{max})$ and $O(m_{alive} \log^d m_{alive})$, respectively.

7. THE WEIGHTED VERSION

We are already very close to solving the most generic form of RTS. The only constraint left is that the element weights are still confined to 1. In this section, we will remove this last constraint.

Weighted Distributed Tracking. Before tackling elements with arbitrary weights, let us first consider a more general version of the DT problem in Section 3.2. As before, the setup involves a coordinator q, and h participants $s_1, s_2, ..., s_h$, such that s_i $(1 \le i \le h)$ keeps a counter c_i initially set to 0. At every timestamp, still at most one counter gets increased, but the increment can be any positive integer (instead of just 1). The coordinator aims to capture *maturity* subject to

$$\sum_{i=1}^{h} c_i \quad \ge \quad \tau. \tag{10}$$

The efficiency goal now is to achieve two purposes simultaneously:

- Communication cost $O(h \log \tau)$ messages (each one word in length).
- The coordinator and participants perform in total $O(n + h \log \tau)$ CPU time, where n is the total number of counter increases at all sites.

If communication *was* the only concern, the problem can be easily settled with $O(h \log \tau)$ messages as follows. Whenever c_i needs to increase by say w, s_i does so by increasing c_i w times, each time by 1. This conversion essentially transforms the current problem into the unweighted version, thus permitting the application of the same algorithm in Section 3.2.

The drawback of the simple reduction lies in the computation cost: as $O(1)$ time is spent on increasing a counter by 1, the total amount of work done becomes $O(\tau + h \log \tau)$. This can be substantially higher than our goal $O(n + h \log \tau)$ if $\tau \gg n$—this will lead to prohibitively high RTS processing cost when we plug the algorithm into our framework.

The astute reader may wonder why the issue did not arise in Section 3.2. In fact, for the old DT problem, there must be exactly $n = \tau$ counter increases at maturity; therefore, the trivially attainable CPU cost of $O(\tau + h \log \tau)$ is good, precisely the complexity $O(n + h \log \tau)$ that we want. This is no longer true for weighted DT.

Next, we explain how to modify the DT algorithm to suit our purposes. Again, if $\tau \le 6h$, we solve the problem by asking each participant to send all counter changes to q. This requires $O(\tau) = O(h)$ messages.

For $\tau > 6h$, q launches a *round* by informing every s_i $(1 \le i \le h)$ the slack $\lambda = \lfloor \tau/(2h) \rfloor$. The way s_i behaves is a bit different from Section 3.2:

Let $\bar{c}_i$ be an integer that equals 0 at the round's beginning. When $c_i - \bar{c}_i \ge \lambda$:

1. Send a signal to q, and increase $\bar{c}_i$ by λ.
2. If $c_i - \bar{c}_i \ge \lambda$ still holds, repeat Line 1, *unless* q has announced the end of this round.

When the number of signals reaches h, q declares to all sites "the round has finished", and collects their precise counters. Maturity is reported if (10) is satisfied. Otherwise, q decreases τ by $\sum_{i=1}^{h} c_i$; and the next round starts.

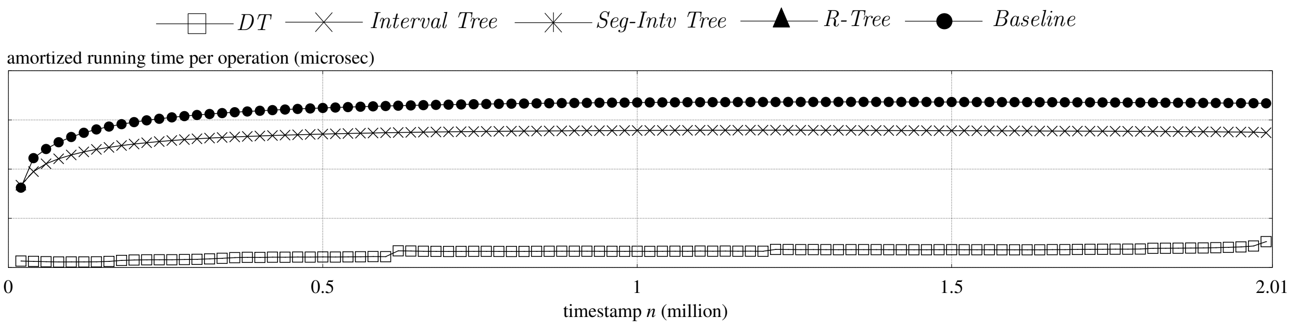
The algorithm is correct because if a round still has not finished at the end of a timestamp, it must hold that

$$\begin{aligned} \sum_{i=1}^{h} c_i \quad &\le \quad \lambda \cdot (h-1) + \sum_{i=1}^{h} (c_i - \bar{c}_i) \\ &< \quad \lambda \cdot h + \lambda \cdot h \le \tau. \end{aligned}$$

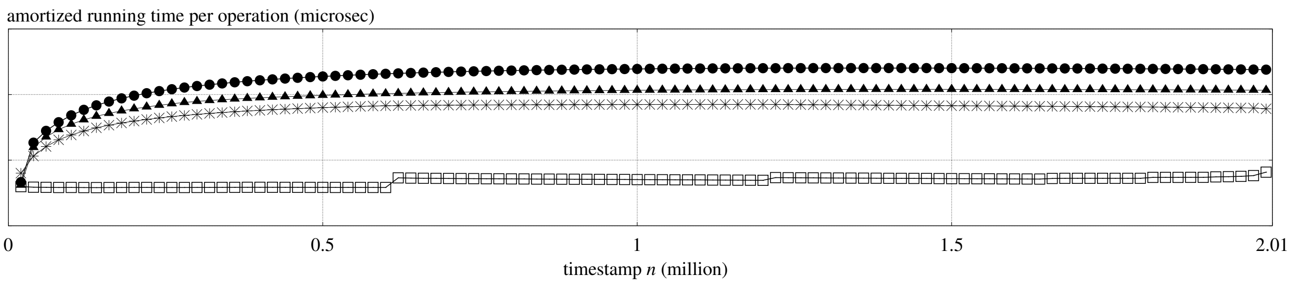
As in Section 3.2, $\tau > 6h$ ensures that a round decreases τ by at least $\tau/3$. Thus, the number of rounds is $O(\log \tau)$, making the total number of messages $O(h \log \tau)$. Furthermore, the computation time is $O(n + h \log \tau)$ because $O(1)$ time is spent (i) at a participant only if its counter is increased or it needs to send a message, and (ii) at the coordinator to send a message.

RTS. Our ultimate RTS algorithm—settling the problem defined in Section 2 entirely—requires only one more sentence to describe: replace the algorithm of Section 3.2 with the above one. All the analysis still holds directly. We thus have established the main result of this paper:

THEOREM 1. *For the RTS problem in constant dimensionality, there is an algorithm that processes n stream elements and m queries in $\tilde{O}(n + m)$ time.*



(a) 1D



(b) 2D

Figure 3: Efficiency as a function of time (m = 1 million, τ = 20 million, static queries)

8. EXPERIMENTS

This section presents an experimental evaluation of our techniques for the RTS problem in 1D and 2D spaces, using the existing solutions for comparison. Specifically, we examined the following methods:

- [1D, 2D] *Distributed tracking* (DT): The proposed algorithm in Theorem 1. The name reflects the new connection we established between RTS and the distributed tracking problem; see Section 4. As explained in Section 5, its time complexity for processing the whole stream is $O(n \log^{d+1} m + m \log^{d+1} m \cdot \log \tau_{max})$.
- [1D, 2D] *Baseline*: The approach described in Section 3.1 that probes all alive queries upon receiving a new element. Its time complexity is $O(nm)$.
- [1D] *Interval tree*: This is the stabbing approach described in Section 3.1 for 1D space, using the interval tree [12] as the stabbing structure. Its time complexity is $\tilde{O}(n) + O(m \cdot \tau_{max})$, as mentioned in Section 3.1.
- [2D] *Seg-Intv tree*: Again the stabbing approach in Section 3.1, but for 2D space, whose stabbing structure combines the segment tree and the interval tree [12]. Its time complexity is also $\tilde{O}(n) + O(m \cdot \tau_{max})$.
- [2D] *R-tree*: As the R-tree is commonly believed to offer competitive efficiency on multidimensional queries, we included another 2D stabbing approach that uses this access method as the stabbing structure. Its time complexity is $O(nm)$—the R-tree is a heuristic structure that does not have attractive efficiency guarantees.

Our machine was equipped with a 3.7GHz multi-core CPU and 16GB memory. The operating system was Linux (Ubuntu 14.04).

We separated the evaluation into two parts: inspecting first the scenario where all queries were registered at the beginning of the stream (Section 8.1), before moving to the second where queries were issued throughout the stream (Section 8.2). The former part studied how the intrinsic parameters m and τ of the RTS problem influence the behavior of alternative algorithms, without the complication caused by the dynamism of arbitrary query registration, the exploration of which was the aim of the latter part.

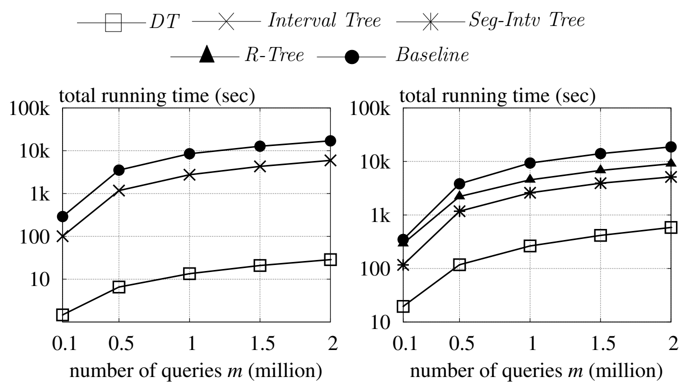
8.1 Scenario 1: Static Query Insertions

Setup. On each dimension, the data space had an integer domain of $[0, 10^5]$. For each stream element e, its value $v(e)$ was a d-dimensional point ($d = 1$ or 2) uniformly distributed in the data space, and its weight $w(e)$ followed the Gaussian distribution with mean 100 and standard deviation 15 (if this resulted in $w(e) < 1$, $w(e)$ was re-generated).

In each experiment, m queries were registered at the beginning of the stream, all given an identical threshold τ. The values of m and τ are the two primary varying parameters in this subsection. Each query q also specified a rectangle R_q, which was a square (for $d = 1$, an interval) with volume 10% that of the data space. The center of R_q was generated in such a way that each coordinate followed the Gaussian distribution with mean $5 \cdot 10^4$ and standard deviation set to 15% of the mean. The whole R_q was required to fall within the data space; otherwise, it was re-generated.

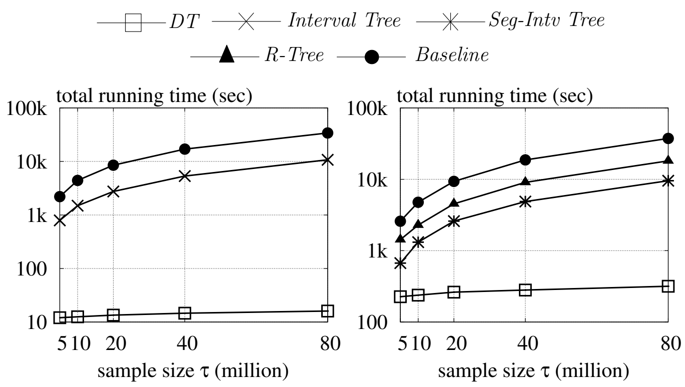
Notice that the generation simulated the situation where stream elements e are everywhere but queries tend to focus on "areas of common interest"—near the center of the data space. The uniformity of $v(e)$ had the importance of ensuring that *every* $v(e)$ should "stab" 10% of the queries currently alive in expectation, regardless of the locations of the query rectangles. This means that each query q had a 10% probability of having its $W(q)$ increased at every timestamp. Given that each increase had the expectation of 100, q was expected to mature after $\tau/(10\% \cdot 100) = \tau/10$ timestamps.

We allowed a query to be terminated before maturity. Specifically, at every timestamp, an alive query q was termi-



(a) 1D (b) 2D

Figure 4: Efficiency as a function of m ($\tau = 20$ million, static queries)



(a) 1D (b) 2D

Figure 5: Efficiency as a function of τ ($m = 1$ million, static queries)

nated with probability p_{del}, whose value was such that the probability for q to be terminated before reaching the $(\tau/10)$-th timestamp was 90%. In other words, 10% of the m queries were able to live till their expected maturity time, echoing the reality where a small fraction of the triggers would get activated eventually. This, intuitively, favored the algorithms falling prey to the quadratic trap because, once deleted, a query can no longer be stabbed by any subsequent elements. Nevertheless, we will see that our algorithm brought forward huge efficiency gains even in this case.

A stream kept evolving until all the queries had either matured or been terminated.

Results. Perhaps the most effective way to illustrate an algorithm's behavior is to *trace* its efficiency for the entire stream. The first set of experiments was designed for this purpose. We used each algorithm to process a 1D/2D stream with m set to 1 million and τ to 20 million, and kept track of the average per-operation cost (where each *operation* refers to the handling of an incoming element, or the insertion, deletion, or maturity of a query) as the stream evolved. Figures 3a and 3b present the 1D and 2D results, respectively.

The comparison reveals the significance of breaking the quadratic trap. The proposed algorithm *DT*—which took only *a fraction of a millisecond* per operation—outperformed all the competitors considerably: by a factor over two and one order of magnitude in 1D and 2D spaces, respectively (note that the y-axis is shown in log scale). Observe that all the other methods required nearly a *millisecond* to perform an operation. This is unacceptable, recalling that one millisecond is comparable to the time of a *disk* I/O, even thought everything was memory resident! The observation serves as strong manifestation of our original motivation that no existing technique can support large-scale RTS on a fast stream.

The figures also reveal several characteristics of the algorithms. At the beginning, each algorithm focused on constructing its underlying data structure, i.e., simply an array for *Baseline*, an endpoint tree for *DT*, and for every other method, the structure as indicated by its name. In any case, the construction cost—after amortization over m queries—was relatively low, compared to the cost of processing an incoming element. This explains the common pattern of all the curves: gradually increasing to a stable value. For *DT*, the occasional "bumps" on its curve indicate global rebuilding (see Section 4), caused by query maturity and deletions.

The next two sets of experiments were designed to examine scalability with respect to m and τ. To do so on m, we fixed τ to 20 million, and measured the total execution time of each algorithm in processing a stream, when the value of m changed from 100k to 2 million. Figures 4a and 4b present the 1D and 2D results, respectively. We investigated the scalability on τ in a similar fashion: fixing m to 1 million, and varying τ from 5 million to 80 million. The results are presented in Figure 5. As predicted by theory, *DT* scaled much better than the other approaches. Its performance advantage became increasingly significant with the growth of m and τ.

8.2 Scenario 2: Dynamic Query Insertions

We now proceed to the scenario where queries can be inserted any time during a stream.

Setup. Stream elements were generated in the same way as in Section 8.1. The total number of elements in a stream was fixed to 3 million. One million queries were registered in the system before the stream started; and then new queries were added according to a mode of dynamism from below:

- *Stochastic mode*: At every timestamp from 1 to 2 million, a new query was registered with probability p_{ins}, which is a parameter to vary in specific experiments.

- *Fixed-load mode*: A new query was registered as soon as an existing one matured or got terminated. In this mode, the number of alive queries remained unchanged during the whole stream.

In any case, a query (regardless of its registration time) was created in the same way as explained in Section 8.1, with its threshold τ set to 20 million. In particular, as before, at every timestamp each alive query had probability p_{del} to be terminated, where p_{del} was set to permit the query to live to its expected maturity time with 10% probability.

Results. Let us start with the stochastic mode. Focusing on $p_{ins} = 0.3$, Figure 6 shows the amortized per-operation cost as a function of the number of received elements. The overall trends, as well as the relative performance of all methods, are similar to those already demonstrated in Figure 3. It is worth pointing out that here the bumps on the curve of *DT*

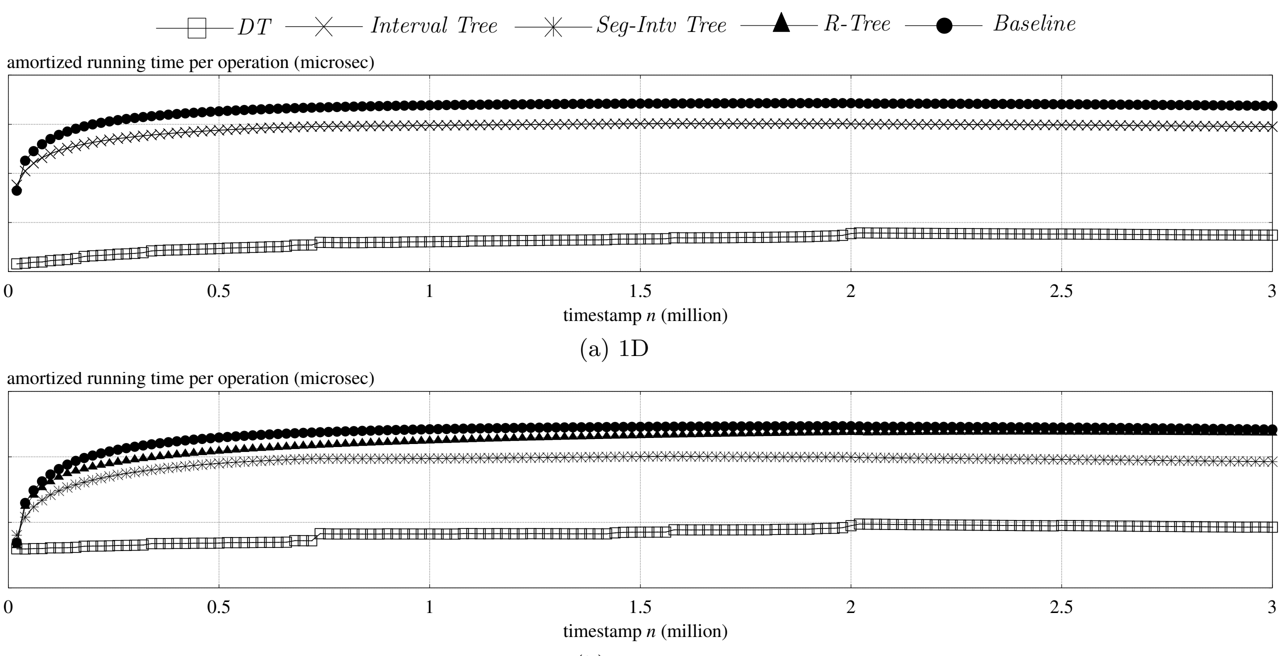


Figure 6: Efficiency as a function of time (dynamic queries, stochastic mode with $p_{ins} = 0.3$)

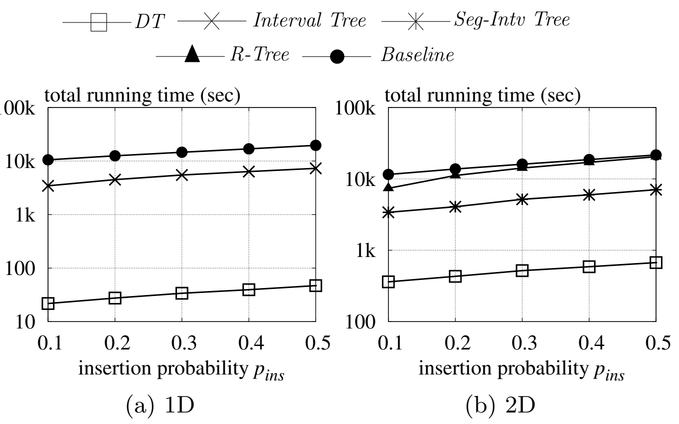


Figure 7: Efficiency as a function of p_{ins} (dynamic queries, stochastic mode)

were the effects of the reconstruction that took place either in global rebuilding or the logarithmic method (Section 5).

Next, we adjusted p_{ins} from 0.1 to 0.5, and measured for each method its total execution time for processing the whole stream. The results are presented in Figure 7. As expected, the running time increased as p_{ins} became larger (which implied more queries)—notice that $p_{ins} = 0.5$ corresponds to a very busy system: one query every two stream elements.

Finally, turning to the fixed-load mode, Figure 8 traces out the amortized per-operation cost of all the methods, confirming once again the significant speedup achieved by our algorithm over the other approaches. The results reflect what performance can be expected in an exceedingly busy system (where the number of alive queries never decreases). Note, interestingly, from Figure 8 that *R-tree* actually performed *worse* even than *Baseline*! This was caused by a well-known drawback of the R-tree's update algorithms: much less effective when the indexed objects have large and heavily overlapping extents. This was exactly the case for RTS (where queries tend to gather in "hot areas"). The problem was most exacerbated on fixed-load streams because they had the highest update volumes—although the astute reader may have also noticed the same phenomenon from Figure 7b.

9. CONCLUSIONS

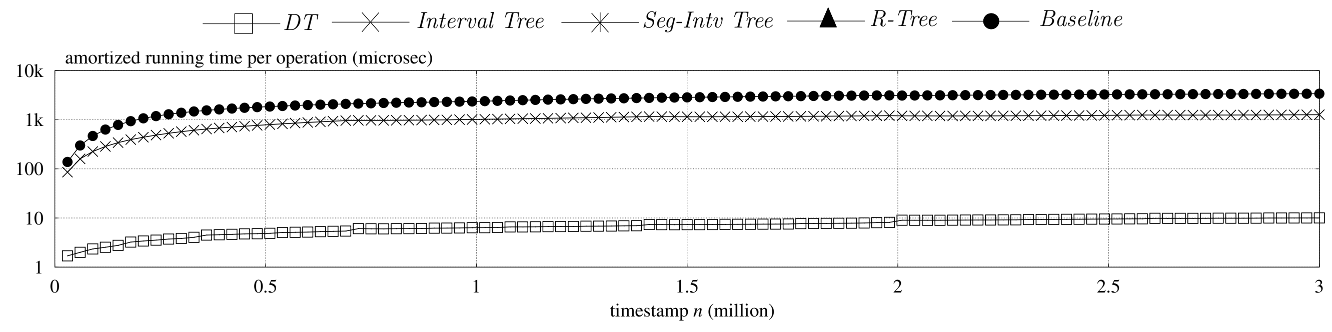
Range thresholding on streams (RTS) supports user triggers of the form "*alert me when a specific number of stream elements have shown up in the region of my interest*". Such kind of triggers are extremely important in numerous system involving time-critical actions. Unfortunately, currently there does not exist an efficient algorithm for processing such triggers: all the known solutions incur prohibitive processing time—quadratic to the problem's input size.

In this paper, we have developed the first algorithm whose running time successfully escapes the quadratic trap—even better, the algorithm promises computation cost that is near-linear to the input size, up to only a polylogarithmic factor. Extensive experimentation has confirmed that the new algorithm, besides having rigorous theoretical guarantees, has excellent performance in practical scenarios, outperforming alternative methods by a factor up to orders of magnitude. It can therefore be immediately leveraged as a reliable fundamental tool for building sophisticated triggering mechanism in real systems.

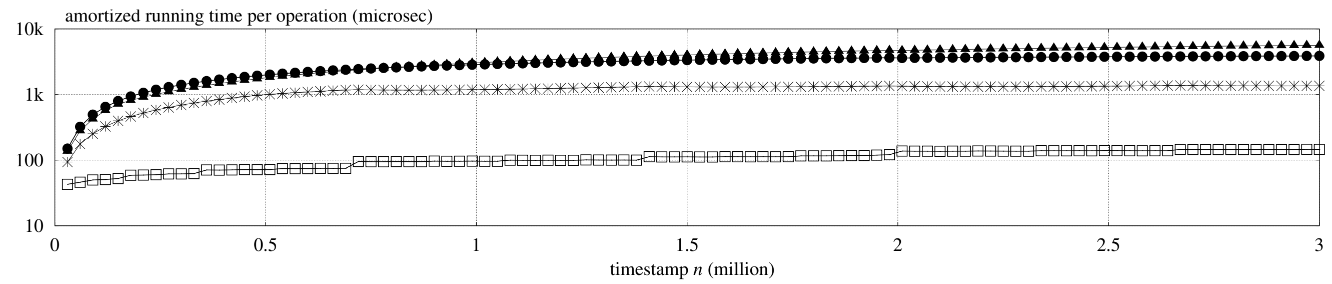
10. REFERENCES

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.

[2] A. Arasu and J. Widom. A denotational semantics for continuous queries over streams and relations. *SIGMOD Record*, 33(3):6–12, 2004.



(a) 1D



(b) 2D

Figure 8: Efficiency as a function of time (dynamic queries, fixed-load mode)

[3] L. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location. *Computational Geometry*, 29(2):147–162, 2004.

[4] L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM J. of Comp.*, 32(6):1488–1508, 2003.

[5] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[6] J. L. Bentley and J. B. Saxe. Decomposable searching problems I: Static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.

[7] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Monitoring streams - A new class of data management applications. In *VLDB*, pages 215–226, 2002.

[8] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *SIGMOD*, pages 379–390, 2000.

[9] G. Cormode, S. Muthukrishnan, and K. Yi. Algorithms for distributed functional monitoring. *ACM Transactions on Algorithms*, 7(2):21, 2011.

[10] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comp. Surv.*, 44(3):15, 2012.

[11] U. Dayal, E. N. Hanson, and J. Widom. Active database systems. In *Modern Database Systems*, pages 434–456. 1995.

[12] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3rd edition, 2008.

[13] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422, 2007.

[14] Y. Diao, S. Rizvi, and M. J. Franklin. Towards an internet-scale XML dissemination service. In *VLDB*, pages 612–623, 2004.

[15] F. Fabret, H. Jacobsen, F. Llirbat, J. L. M. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In *SIGMOD*, pages 115–126, 2001.

[16] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[17] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, pages 49–60, 2002.

[18] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the web. In *SIGMOD*, pages 437–448, 2001.

[19] N. W. Paton and O. Díaz. Active database systems. *ACM Comp. Surv.*, 31(1):63–103, 1999.

[20] S. Rahul. Improved bounds for orthogonal point enclosure query and point location in orthogonal subdivisions in $\mathbb{R}^3$. In *SODA*, pages 200–211, 2015.

[21] D. B. Terry, D. Goldberg, D. A. Nichols, and B. M. Oki. Continuous queries over append-only databases. In *SIGMOD*, pages 321–330, 1992.

[22] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, pages 407–418, 2006.

[23] A. Yu, P. K. Agarwal, and J. Yang. Processing a large number of continuous preference top-k queries. In *SIGMOD*, pages 397–408, 2012.