Approximate Range Thresholding Technical Report

Anonymous Author(s)

ABSTRACT

In this paper, we study the (approximate) Range Thresholding (RT) problem over streams. Each stream element is a d-dimensional point and with a positive integer weight. An RT query q specifies a d-dimensional axis-parallel rectangular range $R(q)$ and a positive integer threshold $\tau(q)$. Once the query q is registered in the system, define $s(q)$ as the total weight of the elements that satisfy: (i) they arrive after q's registration, and (ii) they fall in the range $R(q)$. The task of the system is to capture the *first moment* when $s(q) \geq \tau(q)$. In addition, it admits a more general approximate version: given a real number $0 < \varepsilon < 1$, the task is to capture an arbitrary moment during the period between the first moment when $s(q) \geq (1 - \varepsilon) \cdot \tau(q)$ and the first moment when $s(q) \geq \tau(q)$. The challenge is to support a large number of RT queries simultaneously while achieving *sub-quadratic* overall running time.

We propose a new algorithm called *FastRTS* , which can reduce the exponent in the poly-logarithmic factor of the state-of-the-art *QGT* algorithm from $d + 1$ to d, yet slightly increasing the log term itself. A crucial technique to make this happen is our *bucketing technique*, which eliminates the logarithmic factor caused by the use of heaps in *QGT* algorithm. Moreover, we propose two extremely effective optimization techniques which significantly improve the performance of *FastRTS* by orders of magnitude in terms of both running time and space consumption. Experimental results show that *FastRTS* outperforms the competitors by up to *three* orders of magnitude in both running time and peak memory usage.

ACM Reference Format:
Anonymous Author(s). 2022. Approximate Range Thresholding Technical Report. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In this paper, we study a type of passive queries, called *Range Thresholding* (RT) queries (or *queries* for short when the context is clear). Specifically, we consider a system which receives *elements* from a *data stream*; each element e is represented by a point $v(e)$ in a d-dimensional space with a positive integer weight $w(e)$. The system supports RT queries, where each query q consists of a *query range* that is a d-dimensional *axis-paralleled rectangular range* $R(q)$ and a specified positive integer threshold $\tau(q)$. Once a query q is registered, let $s(q)$ denote the total weight of the elements which (i) arrive after q's registration, and (ii) fall into the query range $R(q)$. Then the task of the system for the RT query q is to capture the *first moment* when $s(q) \geq \tau(q)$. Such a moment is called the *maturity* moment of q, at which q is said to be *matured*.

In addition, RT queries admit a more general approximate version. Given a parameter $0 < \varepsilon < 1$, the task of an *ε-approximate* RT query q is to capture an *arbitrary moment* during the period between the first moment when $s(q) \geq (1 - \varepsilon) \cdot \tau(q)$ and the first moment when $s(q) \geq \tau(q)$. Particularly, when $\varepsilon < \frac{1}{\tau(q)}$, the approximate RT query q is equivalent to capturing the *exact* maturity moment of q.

Applications. The RT queries are especially useful in many scenarios where some time-critical actions must be taken as soon as certain events are detected.

Scenario 1: Stock Trading Systems. A typical application of the RT queries is in the stock trading systems. Each transaction of a stock (e.g., APPL:NSQ) can be considered as a stream element, where the selling (or buying) price is a 1-dimensional point and the number of shares traded in this transaction is the weight. For a query q, its range $R(q)$ can be a *sensitive* price range and the threshold $\tau(q)$ is a specified number of shares of the stock. As soon as a substantial total amount of shares (at least $(1 - \varepsilon) \cdot \tau(q)$) are sold at sensitive prices in $R(q)$ since the registration of q, the system raises an alarm to notify the user. Such a notification is important because a substantial amount of shares sold at some sensitive prices is often an early signal of either a blow-up or a fall of the stock price. Thus, a timely notification is crucial for a user to take timely actions (e.g., either sell or buy the stock) to protect their interests.

Moreover, the above RT query can be easily extended to a multi-dimensional one. For example, a 2-dimensional query may have a form like:

> *Notify me when in total $\tau(q)$ shares (from now) of Apple Inc. (APPL:NSQ) are sold at prices in* [\$140, \$150) *while the price of Alphabet Inc. (GOOG:NSQ) is in* [\$2800, \$2900).

Scenario 2: Public Health Management. During the pandemic, it is important for business to keep track of contact tracing. As such, a shop can register an RT query with a 2-dimensional range around the shop and a threshold of the number of certain close-contact or confirmed-case visiting within the specified range. Each stream element is a visit of either a close-contact or confirmed-case person. As soon as the query is matured, the shop gets a notification to take further actions for safety.

Scenario 3: Bushfire Detection. Likewise, for bushfire detection, an RT query can keep track of the number of abnormal temperature detections from the sensors within a specified area. A larger number of such detections within an area would be an early signal of a bushfire. Therefore, receiving a notification from the system is important to prevent or control the fire as early as possible.

Conventional Solutions. It is not difficult to see that the RT problem is trivial if there is only one query. In this case, one can maintain a *counter* for the query and check for each element e

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

whether $v(e)$ is in the query range. However, the *challenge* lies in supporting a large number of queries simultaneously. Let m be the number of queries and n be the number of elements arrived from the stream so far. For each element, this naive algorithm needs to check and increase counters (if necessary) for m queries. Hence, the time complexity becomes $O(n \cdot m)$, which is *quadratic* and thus, prohibitively expensive since both m and n can be very large.

More sophisticated solutions are to adopt certain data structures, such as the Interval Tree (for $d = 1$) [9, 13], the Segment Tree [13], and the R-tree [5, 17], to efficiently identify those queries whose ranges are "stabbed" by the element e, i.e., $v(e) \in R(q)$, and then increase their counters accordingly. However, since each query q can be stabbed as many as $(1-\varepsilon) \cdot \tau(q)$ times, the time complexity of these algorithms inevitably has the term $O((1-\varepsilon) \cdot \tau_{\max} \cdot m)$, which is another form of *quadratic* term. Here, $\tau_{\max}$ is the largest threshold values among the queries. Unfortunately, none of the above conventional algorithms can overcome these quadratic bounds.

The State of the Art. The *QGT* algorithm [22] (named by the last names of its authors) is the first *sub-quadratic* solution for the RT problem. Its crucial idea stems from an observation on the connection between the RT problem and a seemingly remote problem called the *Distributed Tracking* (DT) [10], where, interestingly, the latter problem is defined in a distributed environment. Based on the DT technique, the *QGT* algorithm achieves an overall time complexity of $O(m \cdot \log^{d+1} m \cdot \log \frac{1}{\varepsilon} + n \cdot \log^{d+1} m)$, and a space consumption bounded by $O(m_{\text{alive}} \cdot \log^d m_{\text{alive}})$ at all time, where m is the number of queries that have ever been registered in the system, n is the total number of elements from the stream so far, and m_{alive} is the number of queries that are still running in the system.

Limitations and Challenges. In theory, both the running time and the space consumption of the *QGT* algorithm are *near-linear* to both m and n, and hence, these bounds are considered as "efficient". However, in practice, just with $\log_2 m \approx 20$, the actual performance of the *QGT* algorithm is already significantly impacted by the *polynomial-logarithmic* factors with exponents $d + 1$ (in the running time) and d (in the space) when $d \geq 3$. As we show later in experiments (in Section 7), the performance of *QGT* deteriorates quickly and becomes worse than some of those aforementioned quadratic competitors for $d \geq 3$. Even worse, on 3-dimensional (rsp. 4-dimensional) datasets just with a moderate size of 2 million (rsp. 1 million) queries, the space consumption of *QGT* quickly exhausts all the available memory (100GB) of our machine! Hence, it fails to complete the corresponding experiment task. This space consumption issue has seriously limited the applicability of the *QGT* algorithm. It turns out that it is still a big challenge to design a sub-quadratic algorithm that is fast and space-efficient in practice.

Our Contributions. We make the following contributions.

A New RT Algorithm. We propose a new algorithm called *FastRTS* for solving the RT problem. Specifically, *FastRTS* achieves an expected overall running time complexity $O(m \cdot \log^d N \cdot \log \frac{1}{\varepsilon} + n \cdot \log^d N)$ and a space consumption bounded by $O(m_{\text{alive}} \cdot \log^d N)$ at all time, where N is the size of the universe $\mathbb{U}$ on each dimension. Comparing to the time complexity of the *QGT* algorithm, *FastRTS* reduces the exponential dependence on dimensionality d for the logarithmic term from $d + 1$ to d, yet slightly increases this term from $\log m$ to $\log N$. Table 1 shows a summary of the complexity comparison.

A Novel Bucketing Technique. We propose a new *Bucketing Technique* which is crucial for *FastRTS* to achieve the theoretical bounds. As we will see in Section 5.2, our bucketing technique allows *FastRTS* to eliminate the $O(\log m)$-factor caused by the use of min-heaps for organizing the DT instances. We note that this bucketing technique is of *independent interests*; it may be able to remove the similar logarithmic factor overhead for all those algorithms which need to organize DT instances with heaps, e.g., the *QGT* algorithm and a very recent work [23].

A New DT Algorithm. In order to facilitate our bucketing technique, we propose a new DT algorithm called *Power-of-Two-Slack DT* (P2S-DT) in Section 5.1. We perform theoretical analysis to show both the correctness and the communication cost of the algorithm. A nice property of our P2S-DT is that all the slack values in this algorithm are power-of-two integers. Thus, our P2S-DT may be of interests in certain scenarios.

Two Effective Optimizations. As the space consumption bound of our *FastRTS* is slightly worse than that of the *QGT* algorithm, to remedy this, we propose two *extremely effective* optimization techniques: (i) the *Range Shrinking* technique (Section 6.1), and (ii) the *Range Counting* technique (Section 6.2). We show that these two techniques not only are theoretically sound, but also dramatically improve the actual performance of *FastRTS* . According to the experimental results in the ablation study, these two techniques improve both the running time and space consumption of our *FastRTS* by up to *three* orders of magnitude.

Extensive Experiments. We conduct extensive experiments on both synthetic datasets and two real stock trading datasets. Experimental results show that our *FastRTS* outperforms all the state-of-the-art competitors by orders of magnitude in both overall running time and space consumption.

2 PRELIMINARIES

We first formally define the *Range Thresholding* (RT) problem and then introduce the *Distributed Tracking* technique and the state-of-the-art *QGT* algorithm for solving the RT problem. These preliminaries will ease the understanding on our proposed techniques.

2.1 The RT Problem Formulation

Let $\mathbb{U} = \{0, 1, 2, \ldots, N-1\}$ be a finite consecutive integer domain, whose size is $|\mathbb{U}| = N$, and where each integer in $\mathbb{U}$ can be encoded with $O(1)$ *words*. All the integers considered in this paper are $O(1)$-word representable. Consider a system which receives data *elements* from a *stream* S and supports the *range thresholding queries*.

The Stream and Elements. The stream S is a sequence of elements $e_1, e_2, \cdots$; the i-th element e_i arrives at *time stamp* i, where $i = 1, 2, \cdots$. In particular, at time stamp 0, the stream S is empty: no element has arrived yet. Each element e_i consists of two fields:

- a *value*, denoted by $v(e_i)$, which is a d-dimensional point in $\mathbb{U}^d$;
- a *weight*, denoted by $w(e_i)$, which is a *positive integer*.

Range Thresholding Queries. Each (range thresholding) query q can be registered in the system at any time stamp *immediately after* the arrival of the corresponding element. Specifically, a query is composed of two fields:

Table 1: A summary of the complexity comparison, where the meaning of the notations can be found in Table 2.

Algorithms	Elem. & Query Dyn. Cost	Range Thresholding Cost	Overall Running Time Cost	Space Consumption
FastRTS	$O(n \cdot \log^d N)$	$O(m \cdot \log^d N \cdot \log \frac{1}{\varepsilon})$ expected	$O(m \cdot \log^d N \cdot \log \frac{1}{\varepsilon} + n \cdot \log^d N)$ expected	$O(m_{\text{alive}} \cdot \log^d N)$
QGT [22]	$O(n \cdot \log^{d+1} m)$	$O(m \cdot \log^{d+1} m \cdot \log \frac{1}{\varepsilon})$	$O(m \cdot \log^{d+1} m \cdot \log \frac{1}{\varepsilon} + n \cdot \log^{d+1} m)$	$O(m_{\text{alive}} \cdot \log^d m)$
Rtree [5, 17]	$O(n \cdot m)$	$O(m \cdot (1-\varepsilon) \cdot \tau_{\max})$	$O(n \cdot m + m \cdot (1-\varepsilon) \cdot \tau_{\max})$	$O(m_{\text{alive}})$
SegInvTree [13]	$O(n \cdot \log^d m)$	$O(m \cdot (1-\varepsilon) \cdot \tau_{\max})$	$O(m \cdot (1-\varepsilon) \cdot \tau_{\max} + n \cdot \log^d m)$	$O(m_{\text{alive}} \cdot \log^{d-1} m)$

- a *range*, denoted by $R(q) = [a_1, b_1) \times [a_2, b_2) \times \cdots [a_d, b_d)$, which is a d-dimensional axis-parallel rectangular range in $\mathbb{U}^d$, where a_i and b_i are integers and $a_i < b_i$ for all $i = 1, 2, \cdots, d$;
- a *threshold*, denoted by $\tau(q)$, which is a *positive integer*.

Furthermore, $R_j(q) = [a_j, b_j)$ is defined as the projection of $R(q)$ on the j^{th} dimension, where $j = 1, 2, \ldots, d$. When the context is clear, a query q and its range $R(q)$ are used interchangeably.

ε-Maturity. Consider a query q *registered* in the system at time stamp t; we define $S(q) \subseteq S$ as the set of all the elements in the stream so far that: (i) arrive since q's registration, and (ii) have values falling in the range $R(q)$. Given a query q and a real number $0 < \varepsilon < 1$, the task of the system is to *raise an alarm* during the period of time between the *first moment* when the total weight of the elements in $S(q)$ becomes *no less* than $(1-\varepsilon) \cdot \tau(q)$, i.e., $\sum_{e \in S(q)} w(e) \geq (1-\varepsilon) \cdot \tau(q)$, and the *first moment* when $\sum_{e \in S(q)} w(e) \geq \tau(q)$. Such a time period is called *ε-maturity period* of q. Any time stamp in the ε-maturity period of q is called an *ε-maturity moment* of q. If a query q is not ε-matured yet, then q is called an *alive* query.

Problem Statement. Given a stream S of elements and a real number $0 < \varepsilon < 1$, the task of the *Range Thresholding problem* is to design an algorithm for the system such that: (i) it supports dynamic query registrations and terminations, and (ii) it correctly captures an *arbitrary* one of the ε-maturity moments for every alive query. The goal is to minimize both the overall running time and the space consumption.

About ε. Although in the above problem statement, the parameter ε is specified to be the same for all queries, each query can actually have a dedicated ε. In particular, by setting $\varepsilon < 1/\tau(q)$, the system can *exactly* capture the first moment t^* for the query q.

2.2 Loosely Related Work

In addition to the conventional and the state-of-the-art solutions to the RT problem that we introduce in the next paragraph, We hereby describe a range of (sub-)fields that are loosely related to our problem.

The first one is *triggers* in DBMS, which can be viewed as a form of RT in concept. Triggers were primarily designed to ensure certain integrity constraints on underlying relations. Subsequently, more complicated forms of triggers were introduced to enable a DBMS to activate itself in response to a greater variety of events, and many efforts have been made to implement such triggers in an efficient manner, falling in the topic of active databases [21].

The second field within which the RT problem might fall is *continuous query processing* over data streams [2, 3, 8, 19]. Essentially, a continuous query is a query that is issued once over a database, and then logically runs continuously over the data in database until it is terminated. When mapping it to the RT problem, the user issuing an RT query should get an alert at the query's maturity time.

The third sub-field is a variant of triggers on streaming data, called *publish/subscribe system*. In this context, users can specify their particular interest via a subscription query (e.g. keywords, tags), such that whenever new data elements flow through the system (e.g. tweets, products, promotions, news articles, etc.), only those elements that are "relevant" to a user's subscription will be "pushed" to the user instantly. In this way, users can alleviate themselves from being overwhelmed in the era of information explosion. Interested readers can refer to [8, 15, 16, 18, 20, 25] for some representative work. To this end, we find RT can be viewed as a form of subscription but even so, it is a new type of subscription query with its unique computational challenge.

The last sub-field is essentially a rejuvenation of the aforementioned active databases on data streams, which yields another line of research under the umbrella of complex event processing [24]. We refer interested readers to some earlier work [12, 14] for more background knowledge. Our work actually complements it in the sense that RT can be treated as another atomic event type that needs to be supported in an efficient and scalable manner.

We note that, unfortunately, none of these related work could solve the RT problem directly or achieve sub-quadratic overall running time.

2.3 Distributed Tracking

The DT problem [11] is defined in a distributed environment, where there are h *participants* u_i ($i = 1, 2, \cdots, h$) and one *coordinator* q. Each participant can only communicate with q by sending and receiving *messages*, each with $O(1)$ *words*. For each participant u_i, there is a *counter* c_i which is 0 initially. The coordinator q has a positive integer *threshold* τ. At each time stamp, *at most* one participant has its counter increased by an arbitrary positive integer value. Given a real number $0 < \varepsilon < 1$, the task of the coordinator is to capture an *arbitrary moment* during the *ε-maturity period* which is the period between the *first moment* when $\sum_{i=1}^{h} c_i \geq (1-\varepsilon) \cdot \tau$ and the *first moment* when $\sum_{i=1}^{h} c_i \geq \tau$. Any such moment during the maturity period is called a *ε-maturity moment*. The goal is to minimize the communication cost: the total number of messages sent and received by the coordinator.

A straightforward solution is to instruct each participant to send a message to the coordinator q when its counter is increased. In this way, q can keep track of the counter sum precisely and report ε-maturity as soon as this sum $\geq (1-\varepsilon) \cdot \tau$. Clearly, the communication cost is $O((1-\varepsilon) \cdot \tau)$ as each counter increment could be 1. When the threshold τ is large, the $O((1-\varepsilon) \cdot \tau)$ communication cost is expensive. A state-of-the-art DT algorithm [10] can achieve a $O(h \cdot \log \frac{1}{\varepsilon})$ communication cost bound. Let $\bar{\tau}$ be the current threshold and initially, $\bar{\tau} \leftarrow \tau$. The DT algorithm works in *rounds*; in each round, it works as follows:

- If $\bar{\tau} \leq 2h$, run the naive algorithm with $O(\bar{\tau}) = O(h)$ communication cost and done;
- Otherwise,
  - q sends a *slack* $\lambda = \lfloor \frac{\bar{\tau}}{2h} \rfloor$ to each participant;
  - for every λ increments on the counter c_i, participant u_i sends a message to q;
  - when q receives the h^{th} message in the current round, q collects the precise counters of all the participants and computes $\tau' = \bar{\tau} - \sum_{i=1}^{h} c_i$; if $\tau' \leq \varepsilon \cdot \tau$, q reports ε-maturity; otherwise, q starts a new round by running a new DT instance with threshold $\bar{\tau} \leftarrow \tau'$ from scratch (with all the counters c_i reset to 0).

Analysis. It can be verified that at the end of each round, τ' is at most a constant fraction of the threshold $\bar{\tau}$, and no more rounds will be performed when $\bar{\tau} \leq 2h$. As a result, there are $O(\log \frac{\tau}{\varepsilon \cdot \tau}) = O(\log \frac{1}{\varepsilon})$ rounds. In each round, the coordinator q receives and sends at most $O(h)$ messages. The total communication cost is thus bounded by $O(h \cdot \log \frac{1}{\varepsilon})$.

Fact 1 ([11]). *There exists an algorithm which solves the DT problem with $O(h \cdot \log \frac{1}{\varepsilon})$ communication cost.*

2.4 The Segment Tree

The next piece of preliminaries is a classic textbook data structure called the *Segment Tree* [13], which is useful for indexing a given set of d-dimensional axis-parallel rectangular ranges, $L = \{\ell_1, \ell_2, \ldots, \ell_m\}$.

The One-Dimensional Case. For the ease of presentation, we start with the case of $d = 1$. Consider a set of 1-dimensional ranges (i.e., intervals) in the universe $\mathbb{U}$, denoted by $L = \{\ell_1, \ell_2, \ldots, \ell_m\}$, where $\ell_i = [a_i, b_i)$ for $i = 1, 2, \ldots, m$. Let $P = \cup_{i=1}^{m} \{a_i, b_i\}$ be the set of the *distinct* endpoints of all the intervals in L.

The Segment Tree on L is essentially an *augmented* complete binary search tree (BST) on P satisfying:

- each endpoint in P is stored in one and exactly one leaf node;
- all the leaf nodes are stored at the same level in ascending order;
- each internal node has exactly two child nodes, except possibly the last one at each level which may have only one child;
- the i^{th} leaf node u is associated with a *left-closed-right-open interval* $R(u) = [v_i, v_{i+1})$, where v_i is the endpoint stored in this leaf node and v_{i+1} is the endpoint stored in the next one, where $i = 1, 2, \ldots, |P| - 1$; for the last leaf node, it is associated with $[v_{|P|}, \infty)$;
- the associated range $R(u)$ of an internal node u is the union of the associated ranges of its child nodes;
- each node u is also associated with a range set $L(u) \subseteq L$ of all the ranges $\ell \in L$ satisfying: (i) the range of u is fully contained in ℓ, i.e., $R(u) \subseteq \ell$, and (ii) the range of u's parent v is not fully contained in ℓ, i.e., $R(v) \not\subseteq \ell$.

Canonical Node Set. For each range $\ell \in L$, the set of all the nodes u that have $\ell \in L(u)$ is defined as the *Canonical Node Set* of ℓ, denoted by $\mathcal{N}(\ell)$. The associated ranges of all the nodes in $\mathcal{N}(\ell)$ are called the *Canonical Sub-ranges* of ℓ, and the set of them is denoted by $\mathcal{C}(\ell)$. By the standard results, $\mathcal{C}(\ell)$ constitutes a *partition* of ℓ. That is, $R(u) \cap R(v) = \emptyset$ for $\forall u, v \in \mathcal{N}(\ell)$ and $\cup_{u \in \mathcal{N}(\ell)} R(u) = \ell$.

Table 2: A summary of frequently used notations

Notations	Descriptions
m	the total number of queries that have ever been registered
n	the total number of elements from the stream so far
m_{alive}	the number of queries that are still running in the system
N	the size of the universe $\mathbb{U}$ on each dimension
ε	the approximation parameter of maturity condition
$\tau(q)$	the threshold of query q
$R(q)$	the range of query q
u, v	tree nodes in the (Incremental) Segment Tree
$\mathcal{N}(q)$	the Canonical Node Set of query q
$L(u)$	the list of all the queries q having node u in $\mathcal{N}(q)$
$\mathcal{B}_i$	a bucket with the slack value 2^i

Example 1. *Figure 1(a) shows a set Q of 2-dimensional queries. Let us focus on the first dimension only. The upper tree in Figure 1(b) is the resulted Segment Tree on Q on the first dimension. The associated range of the leaf node u_1 (highlighted in grey) is $R(u_1) = [3, 4)$ and that of the node u_2 is $R(u_2) = [4, 10)$ which is the union of the associated ranges of its two child nodes. Moreover, the associated range set $L(u_2)$ is $\{q_2, q_5, q_7, q_8\}$ because $R(u_2)$ is fully contained in those query ranges while $R(u_2.parent) = [0, 10)$ is not fully contained in any of them. Furthermore, the Canonical Node Set of q_8 is $\mathcal{N}(R_1(q_8)) = \{u_1, u_2, u_3\}$, where $R_1(q_8) = [3, 17) = [3, 4) \cup [4, 10) \cup [10, 17)$.*

The Multi-Dimensional Case. The above 1-dimensional Segment Tree can be easily extended to $d \geq 2$. The d-dimensional Segment Tree on L has d *layers*, the i^{th} of which corresponds to the i^{th} dimension. Specifically, the 1^{st} layer is a Segment Tree on L on the first dimension only. For each node u in the current layer with $L(u) \neq \emptyset$, a Segment Tree on $L(u)$ on the *next* dimension is constructed at the next layer. Such a construction is repeated until all the d^{th}-layer trees are constructed. For example, Figure 1(b) shows the Segment Trees at the second layer constructed on $L(u_1)$, $L(u_2)$ and $L(u_3)$ on the second dimension, respectively.

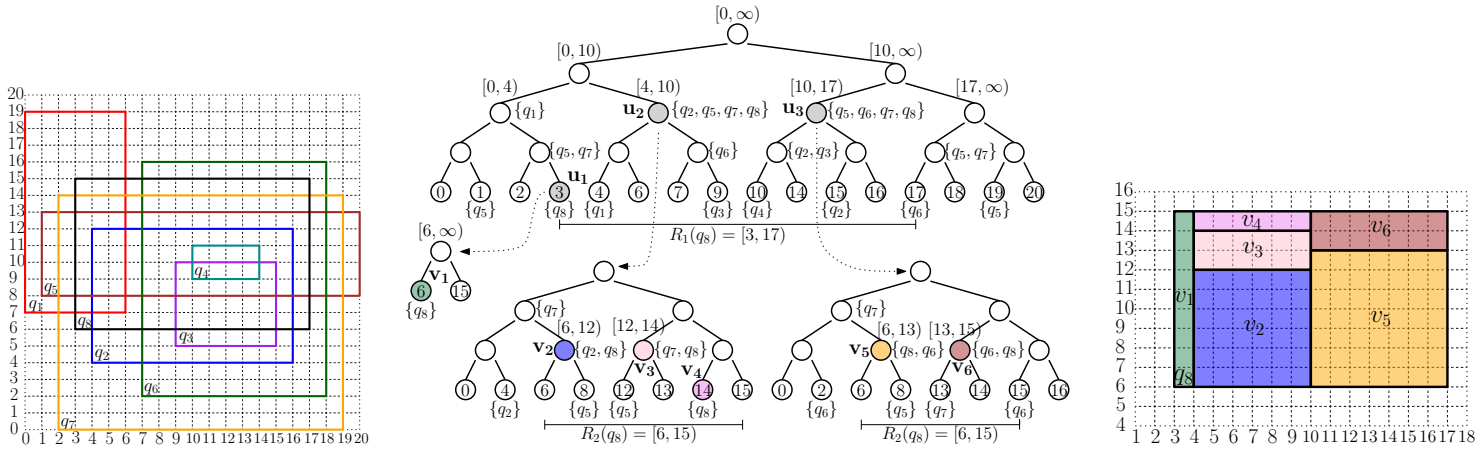
Canonical Node Set. The tree nodes at the d^{th} layer are called as *last-layer* nodes. Each last-layer node u essentially corresponds to a d-dimensional range. With slight abuse of notation, we define this range as the associated range of u, denoted by $R(u)$. For example, for the last-layer node v_1 in Figure 1(b), $R(v_1) = [3, 4) \times [6, 15)$; and $R(v_5) = [10, 17) \times [6, 13)$. For each range $\ell \in L$, the Canonical Node Set is defined as the set of all *last-layer* nodes that have $\ell \in L(u)$. Their associated ranges are the Canonical Sub-ranges of ℓ, the set of which is denoted by $\mathcal{C}(\ell)$. It is known that $\mathcal{C}(\ell)$ forms a *partition* of the range ℓ. Continuing the previous example, $\mathcal{N}(q_8) = \{v_1, v_2, \ldots, v_6\}$, as shown in Figure 1(b). Their associated ranges are shown in Figure 1(c) and constitute a partition of $R(q_8)$.

Finally, we summarize some key results in the following fact:

Fact 2 ([13]). *A d-dimensional Segment Tree on a set of m axis-parallel rectangular ranges L can be constructed in $O(m \cdot \log^d m)$ time and consumes $O(m \cdot \log^d m)$ space. For each range $\ell \in L$, $|\mathcal{N}(\ell)|$ is bounded by $O(\log^d m)$.*

2.5 The State-of-the-Art RT Algorithm

Next, we introduce the *QGT* algorithm [22], a state of the art for solving the RT problem.



(a) A set of query ranges $Q = \{q_1, q_2, \ldots, q_8\}$

(b) The Segment Tree on Q with some secondary trees omitted

(c) The canonical cub-ranges of $R(q_8)$

Figure 1: A 2-Dimensional Segment Tree Example

The One-Time Registration Case. For simplicity, we start with a *restricted case* that all the m queries are registered before the first element in the stream S arrives, i.e., at time stamp 0, and since then, no query registrations are allowed.

The Connection to the Distributed Tracking. The basic idea of the *QGT* algorithm is to construct a d-dimensional Segment Tree $\mathcal{T}$ on (the ranges of) the m queries. As a result, the associated ranges of all nodes in the canonical node set of q, $\mathcal{N}(q)$, form a partition of the range $R(q)$. the range of each query q can be partitioned into a set of canonical sub-ranges $C(q$ A crucial observation in *QGT* is that by maintaining a counter c_u for each last-layer node u, the total weight of those elements that fall in $R(q)$ is equal to the counter sum of all the nodes in $\mathcal{N}(q)$, namely,

$$\sum_{u \in \mathcal{N}(q)} c_u = \sum_{u \in \mathcal{N}(q)} \left(\sum_{e \in S \wedge v(e) \in R(u)} w(e) \right) = \sum_{e \in S \wedge v(e) \in R(q)} w(e) .$$

In other words, the query q and $\mathcal{N}(q)$ constitute a DT instance, denoted by $DT(q)$. Specifically, the query q is the coordinator with $\tau(q)$ as the threshold and each node $u \in \mathcal{N}(q)$ is a participant with c_u as the counter. As a result, an ε-maturity moment of q can be captured by $DT(q)$. Therefore, continuing the previous example, an ε-maturity moment of q_8 can be captured by a DT instance with q_8 as the coordinator and $\tau(q_8)$ as the threshold, and $\{v_1, v_2, \ldots, v_6\}$ as the participants with the counters, respectively.

Organizing DT Participants with Heaps. A last-layer node may involve as a participant in multiple DT instances, e.g., the node v_6 in Figure 1(b) involves in $DT(q_6)$ and $DT(q_8)$. The *QGT* algorithm organizes all the participants of a last-layer node u in the $DT(q)$ for all $q \in L(u)$ with a *min-heap*. Qiao et al. [22] show that, with those min-heaps, all the $DT(q)$ can run properly, yet each "communication" between a participant and its coordinator will be followed by $O(1)$ heap operations. Consequently, an extra $O(\log m)$ factor overhead is introduced in the overall DT running cost.

Running Time Analysis. The overall running time of the one-time registration case consists of two types of costs: (i) the cost for element processing, and (ii) the cost for the DT instances.

First, for each element e, the *QGT* algorithm needs to increase the counter c_u by $w(e)$ for all the last-layer nodes u whose ranges are "*stabbed*" by e, namely, $v(e) \in R(u)$. This can be performed in $O(\log^d m)$ time. Thus, the total element processing cost is bounded by $O(n \cdot \log^d m)$.

Second, for the DT part, by implementing the min-heap of a node u with the standard binary heap [9], each heap operation can be performed in $O(\log |L(u)|) = O(\log m)$ time. The cost of each $DT(q)$ takes $O(|\mathcal{N}(q)| \cdot \log \frac{1}{\varepsilon} \cdot \log m)$ time. By Fact 2, $|\mathcal{N}(q)| = O(\log^d m)$. Therefore, the overall cost is bounded by $O(m \cdot \log^{d+1} m \cdot \log \frac{1}{\varepsilon})$ for m queries.

Putting the above two costs together gives the overall running time cost for the one-time registration case, which is bounded by $O(m \cdot \log^{d+1} m \cdot \log \frac{1}{\varepsilon} + n \cdot \log^d m)$.

Space Analysis. The space consumption is dominated by the size of the Segment Tree $\mathcal{T}$ along with all the DT instances. Thus, this size is bounded by $O(m \cdot \log^d m)$. Moreover, by a careful tree rebuilding strategy (i.e., rebuilding everything if $m_{\text{alive}} < m/2$), the *QGT* algorithm warrants a space consumption bounded by $O(m_{\text{alive}} \cdot \log^d m_{\text{alive}})$ *at all time*, where m_{alive} is the number of queries that are alive in the system.

Supporting Query Dynamics. In order to strengthen the above solution for the one-time registration case to support query registrations and terminations (referred as *query dynamics*), the *QGT* algorithm adopts the standard *logarithmic method* [4, 6], which is a general technique for making a static data structure dynamic. In the case of *QGT* algorithm, the logarithmic method introduces an extra logarithmic factor, i.e., $O(\log m)$, to the element processing cost, and admits an amortized cost of $O(\log^{d+1} m)$ for each query registration and termination, while keeping the space consumption bound unchanged. The final result is summarized below:

Fact 3 ([22]). *For m queries (along with their registrations and terminations) and n elements, the* QGT *algorithm solves the ε-approximate RT problem with:*

- $O(m \cdot \log^{d+1} m \cdot \log \frac{1}{\varepsilon} + n \cdot \log^{d+1} m)$ *overall running time, and*
- $O(m_{alive} \cdot \log^d m_{alive})$ *space consumption at all time, where m_{alive} is the number of alive queries.*

3 AN OVERVIEW OF OUR ALGORITHM

In this section, we give an overview of our proposed algorithm, called *Fast Range Thresholding over Streams* (*FastRTS*), for solving the RT problem. Basically, *FastRTS* adopts the same algorithmic framework of *QGT* .

Comparing to *QGT* , *FastRTS* eliminates:

- the need of the logarithmic method for query dynamics: thus, it removes the logarithmic overhead in element processing and query dynamic;
- the need of heaps for organizing the DT instances: hence, it removes the logarithmic overhead in the DT running cost.

As a result, *FastRTS* improves the overall running time bound of *QGT* by roughly a logarithmic factor.

The *FastRTS* Algorithm. *FastRTS* consists of two main modules: (i) the Query and Element Processing (QEP) Module, and (ii) the DT Manager (DTMgr) Module. Specifically, the QEP module handles both the element processing and query dynamics, while the DTMgr manages the DT instances.

In the next two sections, we introduce the details of these two modules and prove the following theorem:

Theorem 3.1. *For m queries (along with their registrations and terminations) and n elements, the* FastRTS *algorithm solves the ε-approximate RT problem with:*

- $O(m \cdot \log^d N \cdot \log \frac{1}{\varepsilon} + n \cdot \log^d N)$ *overall running time in expectation,*
- $O(m_{alive} \cdot \log^d N)$ *space consumption at all time,*

where N is the size of the universe $\mathbb{U}$ on each dimension.

4 THE QEP MODULE

In this section, we reveal the details of the Query and Element Processing module in *FastRTS* . We first start with a simple idea without worrying about the space consumption. This idea is to construct a *full* Segment Tree on the universe $\mathbb{U}^d$, where the base tree in each layer is on $\mathbb{U}$ of the corresponding dimension. Specifically, we build a Segment Tree on $\mathbb{U}$ for the first dimension, and then for each node, we recursively build a Segment Tree on $\mathbb{U}$ for the next dimension. Figure 2(a) shows a full Segment Tree $\mathcal{T}$ (including all the grey nodes) on $\mathbb{U} = \{0, \ldots, 20\}$. For the 2-dimensional case, each node of $\mathcal{T}$ is associated with a full Segment Tree on the next dimension, as shown in Figure 2(b).

Since the full Segment Tree captures all the possible endpoints on each dimension, the structure of the tree is *static* (i.e., fixed) under query insertions and deletions. Therefore, a query insertion (deletion) is just as simple as to associate (remove) the query to (from) the corresponding canonical nodes in the tree. While the full Segment Tree is efficient for query dynamics, its space consumption can be prohibitive. To see this, there are $O(N)$ nodes in the base tree at the first layer of a full Segment Tree on $\mathbb{U}^d$; each of these nodes has a full Segment Tree on $\mathbb{U}^{d-1}$. As a result, solving the recursion gives the overall space of the d-dimensional base tree bounded by $O(N^d)$.

An Incremental Segment Tree on $\mathbb{U}^d$. We address this issue by constructing a full Segment Tree on $\mathbb{U}^d$ *incrementally*. We call this variant as *Incremental Segment Tree* (*IncSegTree*). The basic idea of the *IncSegTree* is to perform query insertions and deletions as if it is a full Segment Tree, yet a node in *IncSegTree* is materialized only when it is "touched" by some query. For example, in Figure 2 the nodes in black are materialized while those in grey are not. From the standard results [13], a query insertion and deletion can only touch at most $O(\log^d N)$ nodes in a full Segment Tree. Thus, only the same number of nodes in *IncSegTree* will be materialized. Thus, for m query insertions, the space consumption of the *IncSegTree* is bounded by $O(m \cdot \log^d N)$. A node in *IncSegTree* is deleted only when there is no query associated to any node in its sub-tree. We summarize the key results in the following fact.

Fact 4 ([13]). *For a d-dimensional* IncSegTree *on $\mathbb{U}^d$,*

- *each query insertion and deletion takes in $O(\log^d N)$ time;*
- *for each query q, $|\mathcal{N}(q)| = O(\log^d N)$;*
- *each element from the stream can be processed in $O(\log^d N)$ time;*
- *the space consumption is bounded by $O(m_{alive} \cdot \log^d N)$ at all time.*

5 THE DT MANAGER MODULE

We first introduce a new variant of the DT algorithm, namely *Power-of-Two-Slack DT* (P2S-DT). A nice property of P2S-DT is that the slack values are all power-of-two integers. With it, we introduce our *bucketing technique* to organize the DT instances. For each last-layer node u, instead of using a min-heap, the bucketing technique adopts a linked list of *non-empty* buckets, where each bucket stores all the queries in $L(u)$ having the same power-of-two slacks. As we will show in Section 5.2, our bucketing technique can eliminate the $O(\log m)$ factor caused by the min-heaps from the overall DT cost.

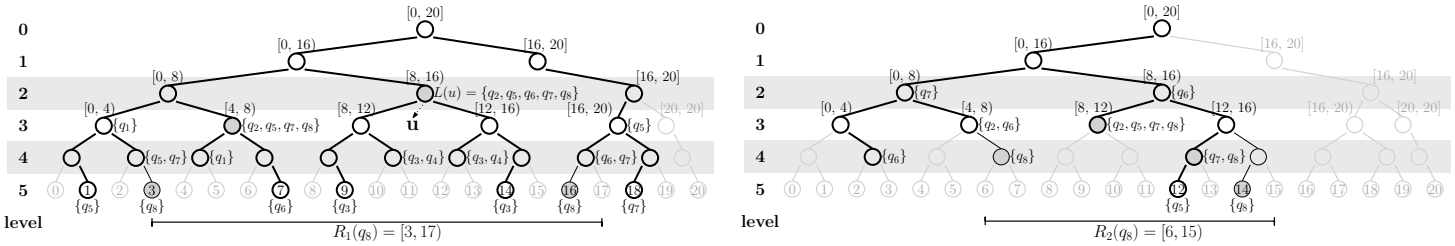
Algorithm 1: The Power-of-Two-Slack DT Algorithm

Data: a coordinator q with a threshold τ and h participants $u_1, \ldots, u_h$: each is with a counter c_i for $i = 1, \ldots, h$
Result: capture an ε-maturity moment for q

```
1  τ̄ ← τ;
2  while true do
3      if τ̄ ≤ 2h then
4          report to q for every counter increment until an ε-maturity
             is captured and then return;
5      for i = 1, ..., h c'_i ← c_i, c̄_i ← c_i, for i = 1, ..., h;
6      q sends a slack λ = 2^⌊log2 τ̄/2h⌋ to each participant;
7      if c_i is increased and ⌊c_i/λ⌋ − ⌊c̄_i/λ⌋ ≥ 1 then
8          u_i sends the counter increment c_i − c̄_i to q;
9          update the last communicated counter: c̄_i ← c_i;
10         if q has ≥ τ̄/2 counter increments since current round then
11             collect counter c_i from u_i for i = 1, ..., h;
12             compute τ' ← τ̄ − (Σ_{i=1}^h c_i − Σ_{i=1}^h c'_i);
13             if τ' ≤ ε · τ then
14                 report ε-maturity and return;
15             else
16                 τ̄ ← τ'; continue, i.e., start a new round;
```

5.1 A New DT Algorithm

To facilitate our bucketing technique, we first develop a new DT algorithm, called *Power-of-Two-Slack DT* (P2S-DT). As shown in Algorithm 1, P2S-DT works in rounds. In each round, the coordinator q sends a *slack* value λ to each participant; whenever a counter c_i



(a) *IncSegTree* materialized by Q on the first dimension

(b) *IncSegTree* materialized by $L(u)$ on the second dimension

Figure 2: Part of the *IncSegTree* on $\mathbb{U}^2 = \{0, \ldots, 20\}^2$ on the query set Q as shown in Figure 1(a)

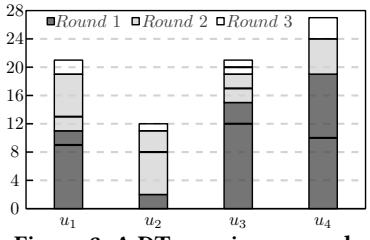


Figure 3: A DT running example

gets increased and "crosses" the next multiple of λ, participant u_i reports the count increment to q. When q receives "enough" counter increments in the current round, it collects the precise counters from all the participants and check ε-maturity. If q matures, then report the maturity; otherwise, q starts a new round.

Example 2. *Figure 3 shows a running example of the P2S-DT algorithm on a DT instance, where the coordinator q has a threshold $\tau = 80$ and there are $h = 4$ participants u_1, u_2, u_3, u_4. Moreover, $\varepsilon = 0.01 < 1/\tau$ means this instance is to capture the exact maturity. In round 1, $c'_i \leftarrow 0$ and $\bar{c}_i \leftarrow 0$ for $i \in \{1, 2, 3, 4\}$, $\tau' \leftarrow \tau = 80$, and slack $\lambda = 2^{\lfloor \log_2 \frac{\tau}{2h} \rfloor} = 8$. As shown in Figure 3, u_1 reports a counter increment 9 to q, u_3 reports 12 and u_4 reports twice: one counter increment is 10 and the other is 9. As a result, q receives in total $40 \geq \tau'/2$ in the current round and hence, q collects the precise counter increments from the four participants and $\tau' \leftarrow \tau' - \sum_i (c_i - c'_i) = 33$. Round 2 starts with $c'_1 = \bar{c}_1 = 11$, $c'_2 = \bar{c}_2 = 2$, $c'_3 = \bar{c}_3 = 15$, $c'_4 = \bar{c}_4 = 19$, and $\lambda = 2^{\lfloor \log_2 \frac{33}{2\times 4} \rfloor} = 4$. Likewise, in this round the total counter increments received by q is $2 + 5 + 6 + 2 + 5 = 20 \geq \tau'/2 = 33/2$. Hence, q collects the precise counter increments and $\tau' \leftarrow \tau' - \sum_i (c_i - c'_i) = 8 \leq 2 \cdot h$. Therefore, in round 3, the algorithm switches to the branch of Line 4 in Algorithm 1, where the participants report every counter increment to q, until the exact maturity of q is captured.*

Next, we prove the correctness of the P2S-DT algorithm and analyse its communication cost.

Lemma 5.1. *During a round before q receives $\bar{\tau}/2$ counter increment, the total counter increment must satisfy: $\sum_{i=1}^{h} c_i - \sum_{i=1}^{h} c'_i < \bar{\tau}$, and hence, q will not miss the ε-maturity period.*

Proof. According to the P2S-DT algorithm, at any time during a round before q receives $\bar{\tau}/2$ counter increment, the total *actual* counter increment is upper bounded by the sum of the counter increment received by q and the maximum possible total counter increment that q is not yet aware of, i.e., $h \cdot (\lambda - 1)$. Thus,

$$\sum_{i=1}^{h} c_i - \sum_{i=1}^{h} c'_i < \frac{\bar{\tau}}{2} + h \cdot (\lambda - 1) < \frac{\bar{\tau}}{2} + h \cdot \frac{\bar{\tau}}{2h} = \bar{\tau}\,.$$

□

Lemma 5.2. *In each round, the coordinator receives at most $3h$ messages from the participants.*

Proof. It suffices to show that after receiving $3h$ messages from the participants, q must be aware of a total counter increment $\geq \bar{\tau}/2$ in the current round. Recall that in P2S-DT, a participant u_i sends a counter increment to q only when c_i reaches or exceeds the λ-multiple successor of $\bar{c}_i$,i.e, $\lfloor \frac{c_i}{\lambda} \rfloor - \lfloor \frac{\bar{c}_i}{\lambda} \rfloor \geq 1$. While in the worst case, the counter increment $c_i - \bar{c}_i$ can be as small as just 1, such worst case can only happen for the *first* report of each participant. Therefore, if a participant u_i has reported $k + 1$ times (for $k \geq 0$), the total counter increment in u_i that q is aware of is at least $k \cdot \lambda + 1$. As a result, for $3h$ reports from the participants, in the worst case, the total counter increment which q is aware of is at least:

$$h \cdot 1 + 2 \cdot h \cdot \lambda = h + 2 \cdot h \cdot 2^{\lfloor \log_2 \frac{\bar{\tau}}{2h} \rfloor} \geq h + 2 \cdot h \cdot \frac{\bar{\tau}}{4h} \geq \frac{\bar{\tau}}{2}\,.$$

□

According to Algorithm 1, the threshold τ decreases by at least half at the end of each round. Thus, there are at most $O(\log \frac{1}{\varepsilon})$ rounds. Moreover, by Lemma 5.2, each round takes $O(h)$ communication cost. Combining Lemma 5.1, we have the following theorem:

Theorem 5.3. *The P2S-DT correctly solves the ε-approximate DT problem with $O(h \cdot \log \frac{1}{\varepsilon})$ communications. When $\varepsilon < \frac{1}{\tau}$, it captures the exact maturity with $O(h \cdot \log \frac{\tau}{h})$ communications.*

5.2 The Bucketing Technique

Let us go back to the context of the RT problem. According to the algorithmic framework of both our *FastRTS* and the *QGT* algorithms, for a last-layer node u in the d-dimensional (incremental) Segment Tree, u serves as a participant in those DT instances $DT(q)$'s, for all $q \in L(u)$, where $L(u)$ is the list of all the queries q having u in the canonical node set $\mathcal{N}(q)$. When u's counter is incremented, u needs to identify all those DT instances, in which the participant of u should notify the corresponding coordinators according to the DT algorithm. In the following, the DT algorithm we consider is P2S-DT, where all the slack values are power-of-two integers.

A Linked List of Buckets. For a last-layer node u, we organize $L(u)$ with a *linked list* of buckets. Specifically, we *conceptually* maintain a dedicated bucket, denoted by $\mathcal{B}_i$, to store all the queries $q \in L(u)$ where the participants of u have a slack value of 2^i in the corresponding DT instances, for each $i \in \{0, 1, \ldots, \lfloor \log_2 \tau_{\max} \rfloor\}$. In particular, we denote the corresponding slack value of a bucket $\mathcal{B}$ as $\lambda(\mathcal{B})$, e.g., $\lambda(\mathcal{B}_i) = 2^i$. However, physically maintaining all these buckets for each last-layer node u may cause an extra $O(\log N)$

overhead in the overall space consumption. Instead, we only maintain a linked list, denoted by $\mathcal{A}(u)$, of all those *non-empty* buckets sorted in ascending order by their corresponding slack values. Furthermore, for each bucket $\mathcal{B}$, we maintain a counter, denoted by $\bar{c}_u(\mathcal{B})$, to record the value of c_u when the queries in $\mathcal{B}$ last communicated with their coordinators; initially, $\bar{c}_u(\mathcal{B})$ is set to the value of c_u when $\mathcal{B}$ is created. Similarly, we also maintain a counter, denoted by $\bar{c}_u(q)$, for each query q to record the value of c_u when the participant of $DT(q)$ last communicated with the coordinator; initially, $\bar{c}_u(q)$ is set to the value of c_u when q is registered.

The Detailed Procedures of the Bucketing Technique.

Procedure for Query Registrations. When a query q is registered, for each node $u \in \mathcal{N}(q)$, perform the following steps:

- find the bucket $\mathcal{B}$ w.r.t. the slack of $DT(q)$ in $\mathcal{A}(u)$;
- if $\mathcal{B}$ does not exist in $\mathcal{A}(u)$,
  - create the bucket $\mathcal{B}$ and insert $\mathcal{B}$ at a *proper position* in $\mathcal{A}(u)$ such that all the buckets are still sorted;
  - initialize $\mathcal{B}$'s last communicated counter value: $\bar{c}_u(\mathcal{B}) \leftarrow c_u$;
- set q's last communicated counter value in u: $\bar{c}_u(q) \leftarrow c_u$;
- store q in $\mathcal{B}$;

Procedure for Query Terminations and Maturity. When a query q is terminated or matured, for each node $u \in \mathcal{N}(q)$,

- locate q in the bucket $\mathcal{B}$ in $\mathcal{A}(u)$;
- remove q from $\mathcal{B}$;
- if $\mathcal{B}$ becomes empty, remove $\mathcal{B}$ from $\mathcal{A}(u)$.

Procedure for Counter Increments. It is shown in Algorithm 2.

Procedure for New Round Starting. The pseudo code is in Algorithm 3.

Algorithm 2: Procedure for Counter Increment

```
Data: c_u is increased from c̄_u to c_u ← c̄_u + w(e)
1  for each bucket B in A(u) do
2      if ⌊c_u/λ(B)⌋ == ⌊c̄_u(B)/λ(B)⌋ then
3          break;
4      if ⌊c_u/λ(B)⌋ − ⌊c̄_u(B)/λ(B)⌋ ≥ 1 then
5          c̄_u(B) ← c_u;
6          for each q ∈ B do
7              send counter increment c_u − c̄_u(q) to q; c̄_u(q) ← c_u ;
8              if DT(q) needs to start a new round then
9                  invoke Algorithm 3;
10             else if DT(q) matures then
11                 invoke Procedure for Query Maturity;
```

Example 3. *Figure 4 shows a running example of the bucketing technique on a node u of range $[8,16) \times [8,12)$ in the* IncSegTree *in Figure 2. This node u serves as participants for the DT instances of the queries q_2, q_5, q_7, q_8. The upper half of Figure 4 shows the status on the bucket list $\mathcal{A}(u)$ before the counter increment $w(e) = 4$. Specifically, at this moment, the counter of u, $c_u = 158$ and there are three buckets $\mathcal{B}_3$, $\mathcal{B}_4$ and $\mathcal{B}_6$. Furthermore, the last communicated counter values for the buckets and queries are as shown. Now, we consider the moment when the counter increment happens, where $c_u = 162$ and $\bar{c}_u = 158$. According to Algorithm 2, it scans $\mathcal{A}(u)$ from the head, i.e., $\mathcal{B}_3$. Since $\lfloor \frac{162}{8} \rfloor - \lfloor \frac{156}{8} \rfloor = 1$, all the queries q in $\mathcal{B}_3$*

Algorithm 3: Procedure for New Round Starting

```
Data: DT(q) starts a new round with a new slack λ = 2^i
1  for each node u ∈ N(q) do
2      find the bucket B in A(u) w.r.t. the previous slack in DT(q);
3      B' ← B;
4      while true do
           // scan the bucket list A(u);
5          if λ(B') == λ then // this implies B_i is found
6              break;
7          if λ(B') < λ then // B_i's predecessor is found
8              create the bucket B_i; insert B_i after B' in A(u);
9              c̄_u(B_i) ← c_u; break;
10         if B' is the head of A(u) and λ(B') > λ then
11             create the bucket B_i; insert B_i before B' in A(u);
12             c̄_u(B_i) ← c_u; break;
13         B' ← B'.prev;
14     remove q from B, and add q to B_i; c̄_u(q) ← c_u;
15     if B becomes empty then
16         remove B from A(u);
```

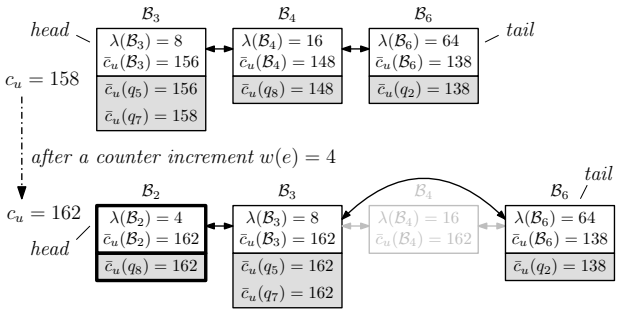


Figure 4: A running example of the Bucketing Technique

report their counter increments $c_u - \bar{c}_u(q)$ to the coordinator and then update their last communicated counter value $\bar{c}_u(q) \leftarrow 162$. Also, the last communicated counter is updated $\bar{c}_u(\mathcal{B}_3) \leftarrow 162$. Then, it scans the next bucket $\lfloor \frac{162}{16} \rfloor - \lfloor \frac{148}{16} \rfloor = 1$. Similarly, the corresponding last communicated counter values are updated, and q_8 reports the counter increment $c_u - \bar{c}_u(q_8) = 10$. But at this time, $DT(q_8)$ needs to start a new round with a new slack $\lambda = 2^2 = 4$. Thus, q_8 should be moved to bucket $\mathcal{B}_2$ and Algorithm 3 is invoked. To find $\mathcal{B}_2$ in $\mathcal{A}(u)$, it scans $\mathcal{A}(u)$ toward the head *from $\mathcal{B}_4$.prev which is $\mathcal{B}_3$. Since $\mathcal{B}_3$ is already the* head *of $\mathcal{A}(u)$ and $\lambda(\mathcal{B}_3) > \lambda$, a new bucket $\mathcal{B}_2$ is created and inserted to $\mathcal{A}(u)$ as a new* head. *(See Line 12 in Algorithm 3). After moving q_8 to $\mathcal{B}_2$, $\mathcal{B}_4$ becomes empty and hence, it is removed from $\mathcal{A}(u)$. This completes the processing for $\mathcal{B}_4$. Since the next bucket $\mathcal{B}_6$ does not meet the reporting criteria, i.e., $\lfloor \frac{162}{64} \rfloor - \lfloor \frac{156}{64} \rfloor = 0$, the procedure for the counter increment returns.*

5.3 Theoretical Analysis

We hereby analyse the time cost of each above individual procedure.

Running Time of Query Registrations. For query registrations, the challenge lies in finding the *successor* non-empty bucket $\mathcal{B}$ in $\mathcal{A}(u)$ for a given power-of-two slack value λ, where $\mathcal{B}$ is the *first* bucket in $\mathcal{A}(u)$ with $\lambda(\mathcal{B}) \geq \lambda$. Next, we prove the following lemma:

Lemma 5.4. *There exists an implementation which can find the successor bucket in $\mathcal{A}(u)$ for any given power-of-two slack in $O(1)$ amortized expected time with $O(|\mathcal{A}(u)|)$ space.*

Proof. In our implementation, there are two main components: (i) a dynamic universal hashing table on the slack-bucket pairs, each of which corresponds to a bucket in $\mathcal{A}(u)$, and (ii) a bit-array which encodes the emptiness status of all the possible buckets, where the $(i+1)^{\text{st}}$ bit is 1 if and only if the bucket $\mathcal{B}_i$ (of slack 2^i) is non-empty, for all $i = \{0, 1, 2, \ldots, \lfloor \log_2 \tau_{\max} \rfloor\}$.

The Hash Table. With the hash table, in $O(1)$ expected time, we can find the pointer of the bucket (if exists) with respect to a given slack value. Moreover, by standard dynamic universal hashing [7], the hash table can achieve: (i) $O(1)$ expected time for each searching, (ii) $O(1)$ amortized time for each slack-bucket pair insertion and (lazy) deletion, and (iii) $O(|\mathcal{A}(u)|)$ space consumption at all time.

The Bit-Array. Without loss of generality, we assume that $\tau_{\max}$ can be represented with one *word*, because it is fairly easy to extend our technique to the case of $O(1)$-word representable $\tau_{\max}$. As a result, the bit-array encoding the emptiness status of all the possible buckets fits in a one-word integer. Thus, the space consumption of this bit-array is $O(1)$.

Suppose the integer value of the bit-array is x; given a target power-of-two slack $\lambda = 2^i$, the corresponding slack value of the successor non-empty bucket (if exists) in $\mathcal{A}(u)$ of λ can be found by the following steps:

- let $y \leftarrow x/2^i$, that is to shift all the lower i bits out;
- if $y = 0$, report the successor bucket of λ does not exist in $\mathcal{A}(u)$;
- otherwise, compute $z \leftarrow y \,\&\, \neg(y-1)$;
- return $z \cdot 2^i$ as the slack value of the successor non-empty bucket of $\lambda = 2^i$ in $\mathcal{A}(u)$.

The correctness of the above calculations follows the fact that $\log_2 z$ is the position of the lowest bit of 1 in y. To see this, suppose the lowest bit of 1 is at position j (starting from 0); $y-1$ flips the j^{th} bit from 1 to 0 and all the bits at positions $< j$ from 0 to 1, while all the bits at positions $> j$ remain unchanged. For example, for $y = 10110\underline{1}00$, the lowest bit of 1 is at position $j = 2$, and $y - 1 = 10110\underline{011}$. As a result, $\neg(y-1)$ (e.g., $\neg(y-1) = 01001\underline{100}$) flips all the lowest j bits back to be the same as those of the original y, yet all those bits at positions $> j$ are flipped to opposite. Therefore, taking the logic-and operation (&) between y and $\neg(y-1)$ gives a binary string z (e.g., $z = 00000\underline{100}$) with 1 at position j and all 0's elsewhere. In other words, $\log_2 z = j$, and the correctness of the above calculations follows. Moreover, all the above calculations can be performed in $O(1)$ time.

Finally, given the slack of the successor bucket $\mathcal{B}$, one can find the pointer of $\mathcal{B}$ with the aforementioned hash table. Putting everything together, the lemma thus follows. □

Thus, by Lemma 5.4, the DT instance for a new query can be constructed in $O(|\mathcal{N}(q)|)$ amortized expected time, linear to the number of participants in this instance.

Running Time of Query Terminations and Maturity. For each query q, by maintaining pointers properly to record the location of query q itself in the bucket $\mathcal{B}$ in each of its canonical node $u \in \mathcal{N}(q)$, locating and removing q from each such bucket can be done in $O(1)$ time. When a bucket becomes empty, the corresponding slack-bucket pair needs to be removed from the hash table, which causes $O(1)$ amortized time. Thus, the running time of each query termination or maturity can be handled in $O(|\mathcal{N}(q)|)$ amortized time.

Running Time of New Round Starting. Consider a query q; when $DT(q)$ needs to start a new round, q needs to be moved from the current bucket to a (possibly new) bucket in each $u \in \mathcal{N}(q)$. As aforementioned, locating q in $\mathcal{B} \in \mathcal{A}(u)$ can be done in $O(1)$ via the pointers. To find the bucket $\mathcal{B}_i$ corresponding to the new slack $\lambda = 2^i$, the procedure scans $\mathcal{A}(u)$ from the current bucket $\mathcal{B}$ towards the head until a "proper position" is found, which is essentially the position of $\mathcal{B}_i$ in (or should be inserted to) the sorted linked list $\mathcal{A}(u)$. Therefore, this cost is bounded by $O(k+1)$, where k is the number of buckets in $\mathcal{A}(u)$ that are "visited" (with slack $\geq \lambda$ but $< \lambda(\mathcal{B})$) and the $O(1)$ term comes from the cost of checking the bucket with slack $< \lambda$ and locating q in $\mathcal{B}$.

Let γ be the total number of rounds of $DT(q)$. Summing over all the γ rounds of $DT(q)$, the total cost of starting new rounds within a node $u \in \mathcal{N}(q)$ is bounded by $O\left((\sum_{r=1}^{\gamma} k_r) + \gamma\right)$, where k_r is the number of buckets visited during the scan of $\mathcal{A}(u)$ when starting the r^{th} round. We claim that $O\left((\sum_{j=1}^{\gamma} k_r) + \gamma\right) = O(\log \frac{1}{\varepsilon})$.

Proof of the Claim. By Theorem 5.3, we know that γ is bounded by $O(\log \frac{1}{\varepsilon})$. Next, we bound $\sum_{j=1}^{\gamma} k_r$. First, observe that the buckets visited during the scan of $\mathcal{A}(u)$ must be the buckets corresponding to the slacks falling in the range of $[\lambda_{\min}, \lambda_{\max}]$, where $\lambda_{\min}$ and $\lambda_{\max}$ are the minimum and maximum possible slacks in $DT(q)$, respectively. Furthermore, according to the P2S-DT algorithm, at the end of each round, the threshold value τ must be decreased by at least a factor of 2, and thus, the slack value in the next round must be at most *half* of the current slack value. In other words, the bucket corresponding to the slack in a new round must be at least "one-step forward" of the bucket $\mathcal{B}$ for the current round in $\mathcal{A}(u)$. As a result, the set of buckets visited during the scan of $\mathcal{A}(u)$ when starting a new different round must be *disjoint*. Therefore, we have:

$$\sum_{r=1}^{\gamma} k_r \leq \log_2 \lambda_{\max} - \log_2 \lambda_{\min} + 1 \leq \log_2 2^{\lfloor \log_2 \frac{\tau}{2h} \rfloor} - \log_2 2^{\lfloor \log_2 \frac{\varepsilon \cdot \tau}{2h} \rfloor} + 1$$
$$\leq \log_2 \frac{\tau/2h}{\varepsilon \cdot \tau/4h} + 1 = O(\log \frac{1}{\varepsilon}) .$$

Note that when a new bucket is created, a new slack-bucket pair is inserted to the aforementioned universal hash table, whose cost is bounded by $O(1)$ amortized. Therefore, by the above claim, the overall maintenance cost of new round starting's in $DT(q)$ is thus bounded by $O(|\mathcal{N}(q)| \cdot \log \frac{1}{\varepsilon})$ amortized.

Running Time of Counter Increments. Next, we analyse the running time bound for handling each counter increment. First, observe that the cost of instructing the queries in the buckets to notify their coordinators can be charged to the "communication cost" of each of the corresponding DT instances. Second, as the costs of handling new round starting's and maturity have been bounded separately, it suffices to bound the total cost of the scan of $\mathcal{A}(u)$. Clearly, the number of buckets that have been visited during the scan of $\mathcal{A}(u)$ for handling a counter increment on c_u is k_u+1, where k_u is the number of buckets that have at least one query to notify the corresponding coordinator, and the "1" is for the first bucket which does not satisfy the notification condition. As a result, the cost of $O(k_u)$ can be charged to the communication cost of those DT instances in the buckets, and the $O(1)$ cost can be charged to

the counter increment, which in turn can be charged to the cost of element processing.

According to Theorem 5.3, the communication cost of $DT(q)$ is bounded by $O(|\mathcal{N}(q)| \cdot \log \frac{1}{\varepsilon})$. By Fact 4, we have $|\mathcal{N}(q)| = O(\log^d N)$ and the total processing cost for n elements is bounded by $O(n \cdot \log^d N)$. Combining the costs of the procedures for query maturity and new round starting, for m queries and n elements, the overall cost of the procedure for counter increments is bounded by $O(m \cdot \log^d N \cdot \log \frac{1}{\varepsilon} + n \cdot \log^d N)$.

Space Consumption. Recall that for each last-layer node u, the data structures for the bucketing technique only take $O(|\mathcal{A}(u)|)$ space. Summing over all such nodes gives the overall space consumption of the DT Manager module bounded by $O(m_{\text{alive}} \cdot \log^d N)$, where m_{alive} is the number of the alive queries.

The Overall Cost of the DT Manager Module. We summarize the key results of the DT Manager module in the theorem below:

Theorem 5.5. *For m queries (along with their registrations and terminations) and n elements, the DT Manager module achieves:*

- $O(m \cdot \log^d N \cdot \log \frac{1}{\varepsilon} + n \cdot \log^d N)$ *time in expectation, and*
- $O(m_{alive} \cdot \log^d N)$ *space consumption at all time.*

Remark. Simultaneously running *overlapped* DT instances that share participants has been found a useful trick in solving various problems, such as in the *QGT* and *FastRTS* for the RT problem, and in a very recent work [23] for the *dynamic structural clustering on graphs*. Both *QGT* and the work [23] adopt binary heaps to organize the DT instances resulting in a logarithmic factor overhead in the DT running cost. Our bucketing technique can be directly applied in these algorithms to substitute the heap-method. Hence, it can immediately improve their overall DT running cost bound by a logarithmic factor. In this sense, our bucketing technique will be of independent interests to other DT-based algorithms for solving other problems. In Section 7.4, we conduct an experiment to study the general practical effects of our bucketing technique, and the results show that it outperforms the heap-method by up to an order of magnitude in terms of running time.

6 EFFECTIVE OPTIMIZATIONS

In this section, we introduce two powerful and effective optimizations for *FastRTS* , which substantially reduce our algorithm's actual running time and the peak memory usage over the entire process. Meanwhile, all the theoretical bounds of *FastRTS* retain.

6.1 The Range Shrinking Technique

In this subsection, we introduce a technique, called *Range Shrinking*, for reducing the number of participants in $DT(q)$ for query q.

The Technique. Consider a query q with range $R(q)$ and threshold $\tau(q)$. The basic idea is to first *extend* q to a *super query* $\tilde{q}$ whose range is a super range of of $R(q)$, namely, $R(q) \subseteq R(\tilde{q})$, and run the DT instance with $\tilde{q}$. The rationale here is that q is ε-matured *only if* the total weight of elements falling in $R(\tilde{q})$ is at least $(1-\varepsilon) \cdot \tau(q)$ since q was registered. As a result, instead of running $DT(q)$ directly, we can first run a DT instance for a super query $\tilde{q}$ with $\mathcal{N}(\tilde{q})$ as participants and with threshold $\tau(\tilde{q}) = \tau(q)$. Based on this observation, indeed, we can run DT instances with respect to a sequence of super queries of q before actually running $DT(q)$. Algorithm 4 shows the pseudo code of this procedure.

Algorithm 4: Procedure for Range Shrinking

Data: $\bar{s}(q)$ is the precise counter sum of all the nodes $u \in \mathcal{N}(q)$ in the *IncSegTree* at the moment when q is registered

initialize a super query $\tilde{q}$ of q with range $R(\tilde{q}) \supseteq R(q)$ and threshold $\tau(\tilde{q}) = \tau(q)$; $i \leftarrow 1$;
while $i \leq \lfloor \log_2 \frac{1}{\varepsilon} \rfloor$ **do**
  $\varepsilon' \leftarrow \varepsilon \cdot \tau(q)/\tau(\tilde{q})$;
  run $DT(\tilde{q})$ to capture an ε'-maturity;
  collect the precise counter sum $s(q)$ of all nodes $u \in \mathcal{N}(q)$;
  $\Delta(q) = s(q) - \bar{s}(q)$; `// get actual counter increments`
  if $\tau(q) - \Delta(q) \leq \varepsilon \cdot \tau(q)$ **then**
    report the ε-maturity of q and return;
  shrink $R(\tilde{q})$ to a smaller range R such that $R(q) \subseteq R \subseteq R(\tilde{q})$;
  update $\tilde{q}$: $R(\tilde{q}) \leftarrow R$ and $\tau(\tilde{q}) \leftarrow \tau(q) - \Delta(q)$; $i \leftarrow i + 1$;
collect the precise counter sum s of all nodes $u \in \mathcal{N}(q)$;
update $\tilde{q}$: $R(\tilde{q}) \leftarrow R(q)$ and $\tau(\tilde{q}) \leftarrow \tau(q) - \Delta(q)$;
run $DT(\tilde{q})$ to capture an ε'-maturity with $\varepsilon' = \varepsilon \cdot \tau(q)/\tau(\tilde{q})$;
report the ε-maturity of q.

Example 4. *Figure 5 shows an example. Consider query q_8 with threshold $\tau(q_8) = 100$ and $\varepsilon = 0.1$. Suppose q_8 is registered immediately after e_2 arrived. As per Algorithm 4, $\bar{s}(q_8) = 40$ because e_1 is the only point in $R(q_8)$ when q_8 was registered. We first run a DT instance for a super query $\tilde{q}_{81}$ with range $R(\tilde{q}_{81}) = [0, 20) \times [0, 16)$, where $\tau(\tilde{q}_{81}) = \tau(q_8) = 100$ and $\varepsilon' = \varepsilon\tau(q_8)/\tau(\tilde{q}_{81}) = 0.1$. When e_6 arrives, $DT(\tilde{q}_{81})$ is ε'-matured, because this is the first moment that the total weight of the points (after q_8's registration) falling in $\tilde{q}_{81}$ is $100 \geq \tau(\tilde{q}_{81})$. However, at this moment, the precise counter sum for q_8 is $s(q_8) = 75$, which means the actual counter increment of $R(q_8)$ is $\Delta(q_8) = s(q_8) - \bar{s}(q_8) = 35$. Thus, $\tau(q_8) - \Delta(q_8) = 65 > \varepsilon \cdot \tau(q_8) = 10$. The algorithm shrinks $\tilde{q}_{81}$ to $\tilde{q}_{82}$ with range $R(\tilde{q}_{82}) = [2, 18) \times [6, 16)$, threshold $\tau(\tilde{q}_{82}) = 65$, and $\varepsilon' = 10/65$. When e_{13} arrives, $\tilde{q}_{82}$ matures and the precise counter sum s (for q_8) is 130 and the actual counter increment of $R(q_8)$ becomes $\Delta(q_8) = 90$. As a result, $\tau(q_8) - \Delta(q_8) = 10 \leq 10$. The algorithm hence reports an ε-maturity of q_8, even without the need of running $DT(q_8)$.*

Correctness. The correctness of the Range Shrinking technique follows from two facts. First, if $DT(\tilde{q})$ is not ε'-matured yet, $DT(q)$ must not be ε-matured. Second, in the for-loop of Algorithm 4, for every ε'-maturity of $DT(\tilde{q})$, a safety check on whether $\tau(q) - \Delta(q) \leq \varepsilon \cdot \tau(q)$ is performed. This ensures that the ε-maturity of q can be reported correctly. Moreover, the last $DT(\tilde{q})$ outside the loop is essentially $DT(q)$ but just tracks the remaining gap between $\tau(q)$ and the actual counter increment in $R(q)$ so far. Thus, an ε-maturity of $DT(q)$ must also be correctly captured in this case.

Running Time Analysis. Denote the super query in the i^{th} loop by $\tilde{q}_i$, and the last super query outside the loop by $\tilde{q}_{\text{last}}$. Observe that the number of rounds in each $DT(\tilde{q}_i)$ is at most $O(\log \frac{1}{\varepsilon})$, because $\tau(\tilde{q}_i) \leq \tau(q)$ always holds, and thus, $\varepsilon' \geq \varepsilon$ holds. Likewise, $DT(\tilde{q}_{\text{last}})$ also has at most $O(\log \frac{1}{\varepsilon})$ rounds. In addition, since $R(\tilde{q}_{\text{last}}) = R(q)$, we have $|\mathcal{N}(\tilde{q}_{\text{last}})| = |\mathcal{N}(q)|$. Therefore, the total running time cost of the DT instances of all these super queries is

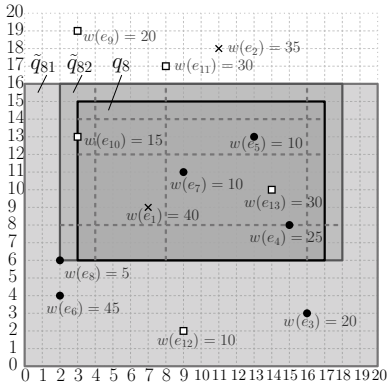


Figure 5: q_8 and its super queries. The elements arrived in the order of their ID's, i.e., e_1 arrives first and e_{13} the last. The weights are shown beside the elements.

bounded by $O\left((\sum_i |\mathcal{N}(\tilde{q}_i)| + |\mathcal{N}(q)|) \cdot \log \frac{1}{\varepsilon}\right)$. Furthermore, each collection of the precise counter sum s for the query q takes $O(|\mathcal{N}(q)|)$ time, and there are at most $O(\log \frac{1}{\varepsilon})$ such collections. Adding up these two costs, the running time of the entire Range Shrinking process is bounded by $O\left((\sum_i |\mathcal{N}(\tilde{q}_i)| + |\mathcal{N}(q)|) \cdot \log \frac{1}{\varepsilon}\right)$.

Next, we show that, by a careful strategy for constructing the super queries, $\sum_i |\mathcal{N}(\tilde{q}_i)|$ can be bounded by $O(\log^d N)$. Therefore, the above overall cost remains $O(\log^d N \cdot \log \frac{1}{\varepsilon})$ as before.

Constructing the Super Queries. To construct a super query $\tilde{q}_i$, we *conceptually* "truncate" all the base trees in *IncSegTree* at the level of 2^i for $i \in \{1, 2, \ldots, \lfloor \log \frac{1}{\varepsilon} \rfloor\}$. Particularly, the *root* is at level-0. We traverse this "truncated" *IncSegTree* as if we compute the canonical node set for q. Specifically, consider a currently visited "leaf" node u (at level-2^i) in a base tree on dimension j. If the associated range of u intersects with $R_j(q)$, the projection of $R(q)$ on the j^{th} dimension, then recurse into the base tree on next dimension if $j < d$ or add u to a temporary set $\mathcal{K}(\tilde{q}_i)$ if this is the last layer, i.e., $j = d$. Finally, we set $R(\tilde{q}_i) = \cup_{u \in \mathcal{K}(\tilde{q}_i)} R(u)$ and then compute the canonical node set $\mathcal{N}(\tilde{q}_i)$ of $\tilde{q}_i$ in the truncated *IncSegTree*.

Example 5. *Figure 5 shows an example of the super queries and their canonical sub-ranges. $\tilde{q}_{81}$ and $\tilde{q}_{82}$ are super queries of q_8 when the truncation level are 2 and 4. To construct the super query $\tilde{q}_{81}$ of q_8 at level 2, the traversal on the base tree on the first dimension in Figure 2(a) stops at the three nodes with ranges $[0, 8)$, $[8, 16)$ and $[16, 20]$. As for the second dimension, the traversal stops at two nodes with ranges $[0, 8)$ and $[8, 16)$ in Figure 2(b). So the $R(\tilde{q}_{81}) = [0, 20) \times [0, 16)$. Likewise, the $R(\tilde{q}_{82}) = [2, 18) \times [6, 16)$. Moreover, as the next shrinking would be at level $2^3 = 8$ which is larger than the height $\lceil \log_2 N \rceil = 5$ of the entire* IncSegTree*, the truncated tree is the tree itself. Thus, the DT instance of the next super query $\tilde{q}_{83}$ degenerates back to $DT(q_8)$ (but with a smaller threshold).*

As per the above construction, three properties hold for all $\tilde{q}_i$: (i) $|\mathcal{N}(\tilde{q}_i)| = O\left((2^i)^d\right)$, (ii) $|\mathcal{N}(\tilde{q}_i)| \leq |\mathcal{N}(q)|$, and (iii) $R(\tilde{q}_i) \supseteq R(q)$. As a result, by (iii), each $\tilde{q}_i$ is a valid super query of q. By (i) and (ii), we have $\sum_i |\mathcal{N}(\tilde{q})_i| = O(\sum_i (2^i)^d) = O(|\mathcal{N}(q)|) = O(\log^d N)$. Therefore, the overall cost of running $DT(q)$ with the Range Shrinking technique is still bounded by $O(\log^d N \cdot \log \frac{1}{\varepsilon})$.

Space Consumption. Since the Range Shrinking process needs to collect the precise counter sum for the query q, the *IncSegTree* still has to materialize $O(|\mathcal{N}(q)|)$ nodes for q to support this operation. Also, the *IncSegTree* needs to materialize $O(|\mathcal{N}(\tilde{q}_i)|)$ nodes for each super query $\tilde{q}_i$. Nonetheless, as per the analysis of the running time, the total number of all these nodes is at most $O(\log^d N)$. The overall space consumption $O(m_{\text{alive}} \cdot \log^d N)$ bound does not change.

Benefits. First, in terms of running time, most counter increments out of $\tau(q)$ are tracked with a considerably smaller DT instance $DT(\tilde{q}_i)$. It also provides a chance of "early termination", in the sense that an ε-maturity of q can be captured even without actually running $DT(q)$. Hence, the actual running time can be substantially reduced. Second, in terms of space consumption, it consists of two main parts: (i) the space of the base tree of *IncSegTree*, and (ii) the space of all the DT instances. As for the base tree, while in theory *IncSegTree* needs to materialize $O(m \cdot \log^d N)$ nodes for m queries, in practice these nodes are largely shared such that the actual number of the nodes materialized is often much smaller. In contrast, the space bound on the total size of all the DT instances, i.e., $O(m \log^d N)$, is pretty solid, as the sizes cannot be shared: one has to store q in each of the nodes in $\mathcal{N}(q)$. Thus, the space consumption is dominated by the total size of the DT instances. Our Range Shrinking technique can keep each DT instance small most of the time. More importantly, it makes the *peak memory usage* of each DT instance *asynchronous*, in the sense that, the DT instances seldom have their maximum sizes, i.e., running with $|\mathcal{N}(q)|$ participants at the same time. Our experimental results show that, *the Range Shrinking technique can effectively reduce both the actual running time and space consumption of* FastRTS .

6.2 The Range Counting Technique

As aforementioned, the Range Shrinking technique allows *FastRTS* to run DT with super queries and thus, reduce the actual space consumption by asynchronizing the peak memory usage of the DT instances. However, the *IncSegTree* still has to materialize $O(|\mathcal{N}(q)|)$ nodes for each query q so as to support the collection of precise counter increment for the range $R(q)$. In other words, even though an ε-maturity of q can be captured by the DT instance of some super query, the nodes of $\mathcal{N}(q)$ still have to be materialized. Therefore, the overall *memory footprint* (i.e., the peak memory usage) may not be desired, especially when the number m of queries is large. Next, we propose an optimization, called the *Range Counting* technique, to address this issue.

The basic idea is to make the *IncSegTree "purely incremental"*. Instead of materializing those $O(|\mathcal{N}(q)|)$ nodes for q when q is registered, it just materializes the nodes for the super query $\tilde{q}_i$ whose DT instance is currently running for q and those nodes for q's previous super queries $\tilde{q}_j$ for all $j < i$. However, the challenge is to support finding precise counter increments in the range $R(q)$.

Our solution to this challenge is to maintain a standard *Range Tree* [13] on the stream elements to support *range counting*'s, which are to report the total weight sum of all the points (in the tree) falling into the given range. Roughly speaking, this Range Tree $\mathcal{T}_r$ is constructed from an empty tree. As each stream element e arrives, insert the point $v(e)$ with weight $w(e)$ into $\mathcal{T}_r$. When $\mathcal{T}_r$ has more than $c \cdot m_{\text{alive}}$ (for some constant $c > 0$) points, destroy $\mathcal{T}_r$ (all the current points are discarded) and maintain $\mathcal{T}_r$ from an

empty tree for the *subsequent* elements. The concrete algorithm works as follows.

- For each query q, maintain two information:
  - $\bar{s}(q)$: the weight sum of the points falling in $R(q)$ in the current Range Tree at the moment when q is registered; if the current Range Tree is constructed (from an empty tree) after q's registration, $\bar{s}(q) = 0$;
  - $s_{\text{prv}}(q)$: the total counter increments for $R(q)$ in the *previous* Range Trees which have been destroyed after q's registration; if there is no such Range Tree, $s_{\text{prv}}(q) = 0$.

  Therefore, the total counter increments happened in $R(q)$ so far can be calculated by $s(q) - \bar{s}(q) + s_{\text{prv}}(q)$, where $s(q)$ is the weight sum of the points in the current Range Tree $\mathcal{T}_r$ falling in $R(q)$.
- When a query q is registered,
  - perform a range counting for $R(q)$ in $\mathcal{T}_r$ to obtain $\bar{s}(q)$, and initialize $s_{\text{prv}}(q) \leftarrow 0$;
  - run $DT(q)$ with the Range Shrinking technique, yet the *IncSegTree* is now purely incremental;
- When a stream element e arrives, invoke Algorithm 5.

Algorithm 5: Procedure for New Stream Element

Data: a stream element e arrives
process e with the *IncSegTree* as before;
for *each* $DT(\tilde{q})$ *that is* ε'*-matured* **do**
  perform a range counting for $R(q)$ to obtain $s(q)$ in $\mathcal{T}$;
  calculate the actual counter increment in $R(q)$ by $\Delta(q) = s(q) - \bar{s}(q) + s_{\text{prv}}(q)$;
  if $\tau(q) - \Delta(q) \le \varepsilon \cdot \tau(q)$ **then**
    report the ε-maturity of q;
  else
    shrink the super query $\tilde{q}$ following Algorithm 4;
insert $v(e)$ with weight $w(e)$ to the current Range Tree $\mathcal{T}$;
if $\mathcal{T}$ *contains more than* $c \cdot m_{alive}$ *points* **then**
  for *each* alive *query* q **do**
    perform a range counting for $R(q)$ to obtain $s(q)$ in $\mathcal{T}$;
    update $s_{\text{prv}}(q) \leftarrow s(q) - \bar{s}(q) + s_{\text{prv}}(q)$; $\bar{s}(q) \leftarrow 0$;
  destroy $\mathcal{T}$ and set $\mathcal{T} \leftarrow \emptyset$;

Example 6. *Let us revisit Example 4. Suppose that the Range Tree $\mathcal{T}_r$ is rebuilt for every $m_{alive} = 8$ elements, i.e., $c = 1$. Now we revisit some special moments. When q_8 is registered, we have $\bar{s}(q_8) = 40$ and $s_{prv}(q_8)$ initialized as 0. When e_6 arrives, the first super query $\tilde{q}_{81}$ (of q_8) matures. At this moment, we need to compute the actual counter increment $\Delta(q_8)$ happened in $R(q_8)$ for safety check (Line 8 in Algorithm 4). Thus, a range counting for $R(q_8)$ is performed to obtain the current counter sum $s(q_8)$ in $\mathcal{T}_r$ and we have $s(q_8) = 75$. As a result, $\Delta(q_8) = s(q_8) - \bar{s}(q_8) + s_{prv}(q_8) = 35$. And then, the DT instance of the next super query, $DT(\tilde{q}_{82})$, starts. When e_8 arrives, the Range Tree $\mathcal{T}_r$ needs to be rebuilt. Specifically, for each alive query q, it performs a range counting for $R(q)$ in $\mathcal{T}_r$, and then updates $s(q)$ and $s_{prv}(q)$ accordingly. As such, for q_8, $s(q_8) = 85$; and then $s_{prv}(q_8) \leftarrow s(q_8) - \bar{s}(q_8) + s_{prv} = 45$ and $\bar{s}(q_8) \leftarrow 0$. When e_{13} arrives, the second super query $\tilde{q}_{82}$ matures; and at this time, in the current Range Tree $\mathcal{T}$, $s(q_8) = 45$. So the actual counter increment is $\Delta(q_8) = 90 \ge (1 - \varepsilon) \cdot \tau$. Hence, q_8 is ε-matured.*

Running Time Analysis. First, observe that when the current Range Tree $\mathcal{T}_r$ is destroyed, the cost of the $O(m_{\text{alive}})$ range counting operations for the alive queries can be charged to the cost of the insertions of the $\Omega(m_{\text{alive}})$ points in $\mathcal{T}_r$. By the standard results [13], each range counting and each point insertion can be performed in $O(\log^d m_{\text{alive}})$ time in a Range Tree with at most $O(m_{\text{alive}})$ points. As a result, each element can be processed in the Range Tree $\mathcal{T}_r$ in $O(\log^d m_{\text{alive}}) = O(\log^d N)$ amortized time. Moreover, by the fact that the processing cost for each element in the *IncSegTree* is bounded by $O(\log^d N)$, the maintenance of the Range Tree does not affect the overall running time bound as before.

Space Consumption. By the standard results [13], it is known that the space consumption of the Range Tree is bounded by $O(m_{\text{alive}} \cdot \log^{d-1} m_{\text{alive}})$. Furthermore, as the *IncSegTree* is now purely incremental, the overall space consumption bound can be written as $O(\sum_{\text{alive query } q} |\mathcal{N}(\tilde{q})| + m_{\text{alive}} \cdot \log^{d-1} m_{\text{alive}})$. The worst-case bound $O(m_{\text{alive}} \cdot \log^d N)$ still holds.

Benefits. By the above Range Counting technique, the space consumption now is just the total size of all the DT instances for the super queries which are run so far plus the space of the Range Tree. Therefore, this space consumption can be substantially smaller than the space of the previous *IncSegTree*. As shown in our ablation study in Section 7.3, this technique can *significantly improve the performance of* FastRTS *by orders of magnitude in terms of both overall running time and the memory footprint.*

7 EXPERIMENTS

The performance evaluation of our *FastRTS* consists of three parts. First, we conduct comprehensive experiments on synthetic datasets with dimensionality $d = 1, 2, 3, 4$. Second, we run experiments on real stock trading data. Last, we perform an ablation study on our proposed optimization techniques, and our bucketing technique.

Competitors. We compare the following methods. All methods are implemented in C++ and complied by gcc 9.3.0 with flag O3 turned on. The source code is at [1].

- ***FastRTS*** : Our *FastRTS* equipped with both the Range Shrinking and Range Counting techniques.
- ***QGT*** : the state-of-the-art *QGT* algorithm.
- ***SegInv***: a conventional stabbing-based approach with a Segment-Interval tree [13] which is a Segment Tree with an Interval Tree as the base tree on the last dimension.
- ***Rtree***: another conventional stabbing-based approach with an R-tree [5, 17].

Machine and OS. All of the experiments were conducted on a machine equipped with an Intel Xeon(R) W-2145 CPU @ 3.70GHz and 100GB RAM running Ubuntu 20.04.3.

Evaluation Metric. We evaluate the performance of an algorithm by measuring the *overall running time* and the *peak memory usage* (i.e., the memory footprint). A competitor algorithm may not have experimental results in certain diagram with tasks under certain parameter settings. This is because either the algorithm fails to complete the task within 10 hours (i.e., 36000 seconds), or its peak memory usage exceeds 100GB breaking down our machine.

Data Generation. In all the experiments, data are generated as follows, unless specified otherwise in the experiment setup.

Element Generation. For each element e, the value $v(e)$ is a point uniformly at random picked in $\mathbb{U}^d$, and the weight $w(e) = \lfloor x + 0.5 \rfloor$, where x is sampled from the Gaussian distribution with $\mu = 10$ and $\sigma = 1$. If $w(e) \notin \mathbb{U}$, generate $w(e)$ again.

Query Generation. All queries have a same threshold $\tau = m$. The range $R(q)$ of each query q is a hypercube (rsp., an interval when $d = 1$ and a square when $d = 2$) in $\mathbb{U}^d$, whose volume is 1% of the entire data space. That is, the side length of $R(q)$ is $\ell = 0.01^{\frac{1}{d}} N$. Given a d-dimensional point $\vec{z}$ as a *seed*, the center of the range $R(q)$ is generated by a d-dimensional Gaussian distribution with *mean* $\vec{\mu} = \vec{z}$, where all the dimensions are independent and each of them has a same standard variance $\sigma = \frac{0.15\ell}{\sqrt{d}}$. In the experiments for synthetic datasets and ablation study, we set $\vec{z}$ to be the center of the entire space, that is $\vec{z}$ has coordinates all equal to $N/2$. Note that we will have different setting for the seed $\vec{z}$ in the experiments on real data. If a generated range $R(q)$ is not fully contained in $\mathbb{U}^d$, then re-generate $R(q)$ by the same process.

Query Registrations and Terminations. As per the above query generation process, each query q is stabbed by an element e with probability of 1%, and the expectation of $w(e)$ is 10. As a result, in expectation, q will be matured after $\frac{\tau}{1\% \cdot 10} = 10\tau$ elements since its registration. However, at each of these 10τ time stamps, we set q to have probability p to be terminated before its maturity. This probability p is set properly such that the probability that q remains alive for 10τ time stamps without being terminated is 20%. As for query registrations, all the experiments are with a *hot-start* setup, that is, there are m queries registered at time stamp 0 at the beginning. Since then, another m queries will be registered within the first 10τ time stamps. To achieve this, we register these m queries following a *Poisson Process* with an *intensity* of $\frac{m}{10\tau}$. As a result, in total $2m$ queries are registered. To ensure the later registered queries have enough chance to get matured, we set the stream length as $n = 20\tau$.

7.1 Evaluation on Synthetic Data

Parameter Settings. We conduct experiments on synthetic datasets with the following parameter settings, where K= 10^3 and M= 10^6. The default value of each parameter is highlighted in bold. When varying a parameter, all the others are set to their default values.

- $N = |\mathbb{U}| = 10\text{M}$, $\tau = m$, $n = 20\tau$, $d \in \{1, 2, 3, 4\}$
- for $d = 1$ or 2, $m \in \{500\text{K}, 1\text{M}, \mathbf{2M}, 5\text{M}, 10\text{M}\}$
- for $d = 3$, $m \in \{200\text{K}, 500\text{K}, \mathbf{1M}, 2\text{M}, 5\text{M}\}$
- for $d = 4$, $m \in \{100\text{K}, 200\text{K}, \mathbf{500K}, 1\text{M}, 2\text{M}\}$
- $\varepsilon \in \{0.01, 0.02, \mathbf{0.05}, 0.1, 0.2\}$

Comparisons on Overall Running Time. Figures 6 (a) - (d) show the overall running time of all methods when varying m on different dimensionality d, and we have the following observations. First, our *FastRTS* is the only one that can complete all the experiment tasks. Second, over all the four dimensionalities, *FastRTS* consistently outperforms *QGT* by around an order of magnitude. Third, *FastRTS* outperforms the two conventional stabbing-based approaches by up to *three* (for $d = 1$) and *two* (for $d = 2$) orders of magnitude. In particular, while *FastRTS* can complete the running tasks with the default m within 100 seconds for $d = 1$ (rsp., 1000 seconds for $d = 2$), these two competitors require about 10 hours for *SegInv* and even more for *Rtree*. This nicely verifies the superiority of our theoretical running time bound. Furthermore, *QGT* is also up to 10 times faster than these two methods. However, when $d \geq 3$, *Rtree* and *SegInv* start becoming competitive to *FastRTS* . In particular, *FastRTS* even slightly loses to *SegInv* when $d = 4$, for two possible reasons: the dataset sizes are considerably small when $d \geq 3$, as we hope to have more competitors completing the running tasks; with d increasing, the poly-logarithmic gets worse exponentially, which severely impacts the performance of both *FastRTS* and *QGT* .

Furthermore, Figures 6(e) - (h) show the running time when varying ε. From these figures, *FastRTS* clearly outperforms all the other competitors who completed the running tasks when $d \leq 3$. But *FastRTS* is inferior to *SegInv* when $d = 4$. Moreover, as expected, all the four competitors run faster when ε increases.

Comparisons on Peak Memory Usage. As shown in Figure 7, *Rtree* is the most space-efficient because the space complexity of *Rtree* is just $O(m)$. However, as aforementioned, *Rtree* is also the slowest among the four. On the other hand, the peak memory usages of both *SegInv* and *QGT* increase quickly with the dimensionality d increasing; both quickly use up the 100GB memory on relatively small datasets: $d = 3$ with $m = 2\text{M}$ and $d = 4$ with $m = 0.5\text{M}$ for *QGT* and $m = 1\text{M}$ for *SegInv*. In contrast, the peak memory usages of our *FastRTS* on these settings are much more friendly; they are: 5.77GB, 9.11GB and 20GB, respectively. The memory usage trend of *FastRTS* increases much slower than these two algorithms. From Figure 7, on some settings, *FastRTS* outperforms these two algorithms by up to two orders of magnitude.

In summary, in these synthetic experiments, *FastRTS* is much more scalable and stable over all various settings than the others competitors, in terms of both running time and space consumption.

7.2 Evaluations on Real Stock Trading Data

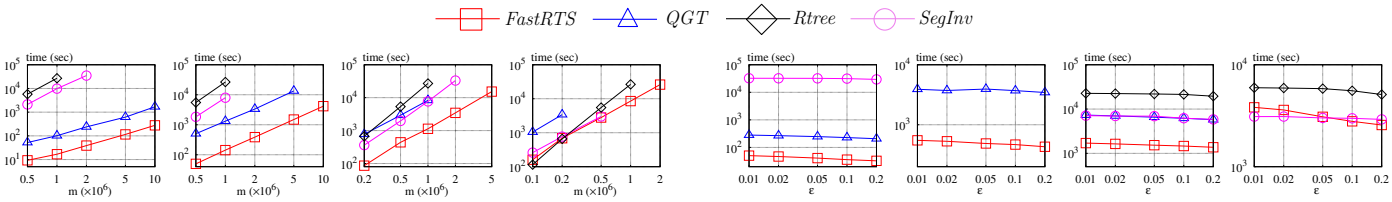
Data Description. We run experiments on the real trading history of two Stocks, A and B. These data [1] are the transactions of Stock A (rsp. Stock B) in 2015-2021. For each transaction of A, we extract a stream element e with $v(e)$ equal to the *price* and $w(e)$ equal to the *volume* (i.e., number of shares). Thus, a 1-dimensional stream w.r.t. the transaction history of A is generated. To generate a 2-dimensional stream, for each element e generated for the transaction of A, we search the *predecessor* transaction of B in terms of time stamp, and extract the price t in this transaction of B as the second dimension value of e. Thus, a 2-dimensional element e' with value $v(e') = (v(e), t)$ and weight $w(e') = w(e)$ is generated.

Query Generation. The queries in this set of experiments are generated in the same way as before. Here, the seed $\vec{z}$ is set as the *average* of all the element points for the first m queries which will be registered at the beginning; $\vec{z}$ is set as the i^{th} element point if a query is generated to be registered at time stamp i.

Parameter Settings (on the real stock trading data).

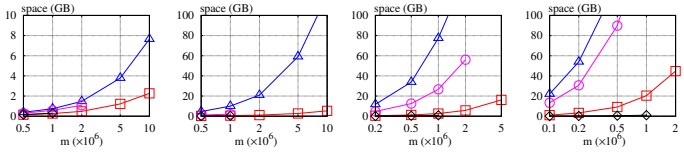
- $N = |\mathbb{U}| = 100\text{K}$, $\tau = m$, $n = 20\tau$
- $d \in \{1, 2\}$
- $m \in \{500\text{K}, 1\text{M}, 2\text{M}, \mathbf{5M}, 10\text{M}\}$
- $\varepsilon \in \{0.01, 0.02, \mathbf{0.05}, 0.1, 0.2\}$

Overall Running Time. Figures 8 (a) and (b) show the results of the overall running time v.s. m, the number of queries on the real datasets. As expected, *FastRTS* is consistently the fastest on both $d = 1$ and $d = 2$. Interestingly, while *QGT* is the second best performer



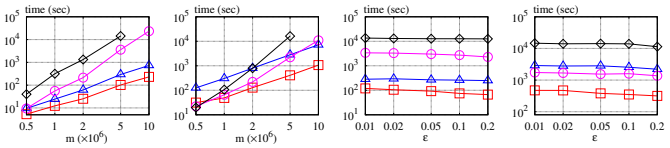
(a) $d = 1$ (b) $d = 2$ (c) $d = 3$ (d) $d = 4$ (e) $d = 1$ (f) $d = 2$ (g) $d = 3$ (h) $d = 4$

Figure 6: Overall running time v.s. m [(a) - (d)] and v.s. ε [(e) - (h)] on synthetic datasets



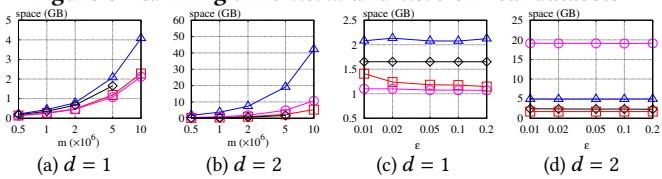
(a) $d = 1$ (b) $d = 2$ (c) $d = 3$ (d) $d = 4$

Figure 7: Peak memory usage v.s. m on synthetic datasets



(a) $d = 1$ (b) $d = 2$ (c) $d = 1$ (d) $d = 2$

Figure 8: Running time v.s. m and v.s. ε on real datasets



(a) $d = 1$ (b) $d = 2$ (c) $d = 1$ (d) $d = 2$

Figure 9: Peak memory usage v.s. m and v.s. ε on real datasets

on $d = 1$, it is outperformed by *SegInv* on $d = 2$. At first glance, this looks inconsistent with the comparisons on the synthetic datasets. We find this is caused by the *element weights*. Although the query thresholds τ's in both experiments have similar values, the average weight of the elements are quite different. Specifically, the average element weight of the real data is about 400; the expected weight of the elements in synthetic datasets is 10. As a result, although the two τ have the same value, the *effective* threshold in the real data is much smaller. Recall that the time complexity of *SegInv* is often dominated by $O(m \cdot \tau)$, so with an effectively much smaller τ, *SegInv*'s performance gets better. Furthermore, from Figures 8 (c) and (d), all the algorithms run faster with larger ε as expected.

Peak Memory Usage. As shown in Figure 9, the peak memory usage comparisons on the real datasets are consistent with those on the synthetic ones on $d = 1$ and $d = 2$ (shown in Figure 7 (a) and (b)). Specifically, the space consumption of the *QGT* algorithm is consistently the largest with a clear gap from those of the other three competitors. Observe that the space consumption of *SegInv* is smaller than that of *Rtree* on $d = 1$, since both the space consumptions of them are linear to m in this case. However, when $d = 2$, this ranking reverses, as the space consumption bound of *Rtree* is not affected by d but that of *SegInv* grows exponentially fast with d. Interestingly, despite of the theoretical space bound which also grows exponentially with d, the peak memory usage of our *FastRTS* is consistently between those of *SegInv* and *Rtree*, and very close to the winner between the two in both the cases of $d = 1$ and $d = 2$. This is because our optimization techniques enable *FastRTS* to avoid the worst case, and hence, perform much better in practice than as the theoretical bound suggests. This clearly shows the effectiveness of our optimization techniques.

7.3 Ablation Study

The last set of experiments is an ablation study on the effectiveness of our optimization techniques. The competitors in these experiments are the variants of *FastRTS* only. In addition to the "fully-geared" *FastRTS* , we also consider (i) *FastRTS-RS* that is equipped with the Range Shrinking technique only, and (ii) *FastRTS-Vanilla* that is the *raw* version of the algorithm.

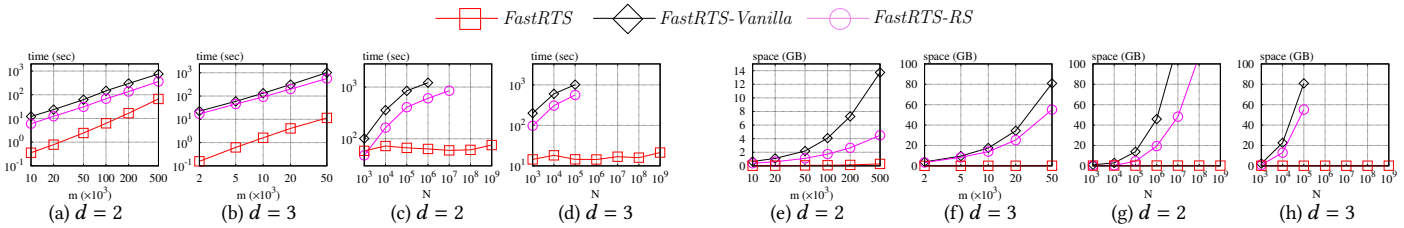
Moreover, it is worth mentioning that in these experiments, we intended to use small m to minimize its impact to the actual performance so as to we could have a clearer view on the impact of the universe size N. The parameter setting is as follows:

- $\tau = m$, $n = 20\tau$, $\varepsilon = 0.05$
- $N \in$ {1K, 10K, **100K**, 1M, 10M, 100M, 1000M}
- for $d = 2$, $m \in$ {10K, 20K, 50K, 100K, 200K, **500K**}
- for $d = 3$, $m \in$ {2K, 5K, 10K, 20K, **50K**, 100K}

Overall Running Time. From Figures 10(a) - (d), with no surprise, the fully-geared *FastRTS* consistently outperforms the other two versions by up to two orders of magnitude, in terms of efficiency. This clearly shows the significance of the two optimization techniques on improving the efficiency. Moreover, *FastRTS-RS* is faster than the raw *FastRTS-Vanilla* on all settings. However, the gap is just not as significant. A possible explanation is that although the Range Shrinking technique allows *FastRTS-RS* to track a considerable portion of counter increments out of $\tau(q)$ with relatively smaller DT instances, materializing all the nodes in $\mathcal{N}(q)$ is still a considerable burden on the overall running time. On the other hand, this is actually a strong evidence of the effectiveness of Range Counting technique. Apart from this, from Figures 10(c) and (d), we can see that the running time of *FastRTS* is not affected by the universe size N as much as suggested by the theoretical bound. Once again, another evidence of the effectiveness of our optimizations.

Peak Memory Usage. As shown in Figures 10(e) - (h), the fully-geared one is always far at the bottom with a dramatic gap from the other two versions. The capability of avoiding materializing the nodes in $\mathcal{N}(q)$ of our Range Counting technique is just powerful. While *FastRTS-RS* is also largely better than the vanilla version on space consumption, for $N = 100$M, it still has to run out of the 100GB memory even just on $m = 50$K when $d = 3$.

From all the above comparisons in the ablation study, the fully-geared *FastRTS* is way better than the other, showing that our optimizations are extremely effective.



(a) $d = 2$ (b) $d = 3$ (c) $d = 2$ (d) $d = 3$ (e) $d = 2$ (f) $d = 3$ (g) $d = 2$ (h) $d = 3$

Figure 10: Ablation Study on running time v.s. m and N [(a) - (d)], and on space v.s. m and N [(e) - (h)]

7.4 Effectiveness of the Bucketing Technique

Last, we conduct an experiment to study the general practical effectiveness of our bucketing technique on running overlapped DT instances simultaneously.

Experiment Setup. In this experiment, we first generate m DT instances, where m is varied in {1K, 10K, 100K, 1M, 10M}. Each DT instance has 100 participants randomly chosen from the total 1000 participants, and the threshold τ is uniformly at random chosen from the integer range [100K, 1M]. The goal is to capture their exact maturity. Furthermore, at each time stamp, we randomly select one of the 1000 participants to increase its counter by a random increment w uniformly chosen from $\{1, 2, \ldots, 31\}$. Whenever a DT instance matures, we generate a new instance in the same way until in total another m DT instances have been generated. We measure the overall running time when all these $2m$ DT instances mature for both our bucketing technique (called *Bucket*) and the heap-method (called *Heap*), respectively.

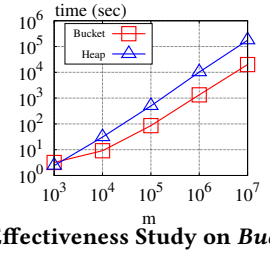


Figure 11: Effectiveness Study on *Bucket* and *Heap*

Figure 11 shows the overall running time of *Bucket* and *Heap* v.s. m. When $m = 1000$, the overall running time of the two methods are almost the same, while with m increasing, the performance gap becomes larger and larger. Especially, when $m = 10^7$, *Bucket* outperforms *Heap* by up to an order of magnitude. This is because, with a larger m, the average number of DT instances that a participant involves becomes larger. Hence, the average length $\bar{L}$ of the heaps in *Heap* is larger, making each heap operation become more expensive. Recall that the overall cost of *Heap* is bounded by $O(cost_{\text{dt}} \cdot \log \bar{L})$, while the overall cost of *Bucket* is $O(cost_{\text{dt}})$, where $cost_{\text{dt}}$ is the total "communication cost" of all the DT instances. Therefore, it clearly shows the superiority of our bucketing technique over *Heap* in running overlapped DT instances simultaneously. On the other hand, as aforementioned, running overlapped DT instances simultaneously has been found useful for solving various problems. The general applicability of our bucketing technique is just as significant.

8 CONCLUSION

In this paper, we propose a new algorithm, called *FastRTS* , for solving the approximate Range Thresholding (RT) problem. In theory, *FastRTS* improves the state of the art by reducing the exponential dependence on the data dimensionality d for the logarithmic factor, yet with a sacrifice of slightly increasing the logarithmic term. The crucial idea to make this improvement is our *bucketing technique*, which helps remove the logarithmic overhead caused by the use of heaps. Our bucketing technique is also applicable to eliminate a logarithmic overhead for all those algorithms in similar scenarios. We further propose two extremely effective optimizations to significantly boost the performance of *FastRTS* . Experimental results show that *FastRTS* outperforms the competitors by orders or magnitude in terms of both overall running time and space consumption.

REFERENCES

[1] [n.d.]. FastRTS source code and experiment dataset. https://anonymous.4open.science/r/FastRTS.

[2] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. 2003. Aurora: a new model and architecture for data stream management. *The VLDB Journal* 12, 2 (2003), 120–139.

[3] Arvind Arasu and Jennifer Widom. 2004. A Denotational Semantics for Continuous Queries over Streams and Relations. *SIGMOD Record* 33, 3 (2004), 6–12.

[4] Lars Arge and Jan Vahrenhold. 2004. I/O-efficient dynamic planar point location. *Comput. Geom.* 29, 2 (2004), 147–162. https://doi.org/10.1016/j.comgeo.2003.04.001

[5] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990*, Hector Garcia-Molina and H. V. Jagadish (Eds.). ACM Press, 322–331. https://doi.org/10.1145/93597.98741

[6] Jon Louis Bentley and James B. Saxe. 1980. Decomposable Searching Problems I: Static-to-Dynamic Transformation. *J. Algorithms* 1, 4 (1980), 301–358. https://doi.org/10.1016/0196-6774(80)90015-2

[7] Larry Carter and Mark N. Wegman. 1979. Universal Classes of Hash Functions. *J. Comput. Syst. Sci.* 18, 2 (1979), 143–154. https://doi.org/10.1016/0022-0000(79)90044-8

[8] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. 2000. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of ACM Management of Data (SIGMOD)*. 379–390.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition.* MIT Press. http://mitpress.mit.edu/books/introduction-algorithms

[10] Graham Cormode, S. Muthukrishnan, and Ke Yi. 2011. Algorithms for distributed functional monitoring. *ACM Trans. Algorithms* 7, 2 (2011), 21:1–21:20. https://doi.org/10.1145/1921659.1921667

[11] Graham Cormode, S. Muthukrishnan, and Ke Yi. 2011. Algorithms for Distributed Functional Monitoring. *ACM Trans. Algorithms* 7, 2, Article 21 (March 2011), 21:1–21:20 pages.

[12] Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *Comput. Surveys* 44, 3 (2012), 15.

[13] Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. 2008. *Computational geometry: algorithms and applications, 3rd Edition.* Springer. https://www.worldcat.org/oclc/227584184

[14] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. 2007. Cayuga: A General Purpose Event Monitoring System. In *Proceedings of Biennial Conference on Innovative Data Systems Research (CIDR)*. 412–422.

[15] Yanlei Diao, Shariq Rizvi, and Michael J. Franklin. 2004. Towards an Internet-Scale XML Dissemination Service. In *Proceedings of Very Large Data Bases (VLDB)*. 612–623.

[16] Françoise Fabret, Hans-Arno Jacobsen, François Llirbat, João L. M. Pereira, Kenneth A. Ross, and Dennis Shasha. 2001. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe. In *Proceedings of ACM Management of Data (SIGMOD)*. 115–126.

[17] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, Beatrice Yormark (Ed.). ACM Press, 47–57. https://doi.org/10.1145/602259.602266

[18] Yuchen Li, Zhifeng Bao, Guoliang Li, and Kian-Lee Tan. 2015. Real time personalized search on social networks. In *ICDE*. 639–650.

[19] Samuel Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. 2002. Continuously adaptive continuous queries over streams. In *Proceedings of ACM Management of Data (SIGMOD)*. 49–60.

[20] Benjamin Nguyen, Serge Abiteboul, Gregory Cobena, and Mihai Preda. 2001. Monitoring XML Data on the Web. In *Proceedings of ACM Management of Data (SIGMOD)*. 437–448.

[21] Norman W. Paton and Oscar Díaz. 1999. Active Database Systems. *Comput. Surveys* 31, 1 (1999), 63–103.

[22] Miao Qiao, Junhao Gan, and Yufei Tao. 2016. Range Thresholding on Streams. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 571–582. https://doi.org/10.1145/2882903.2915965

[23] Boyu Ruan, Junhao Gan, Hao Wu, and Anthony Wirth. 2021. Dynamic Structural Clustering on Graphs. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1491–1503. https://doi.org/10.1145/3448016.3452828

[24] Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-performance complex event processing over streams. In *Proceedings of ACM Management of Data (SIGMOD)*. 407–418.

[25] Albert Yu, Pankaj K. Agarwal, and Jun Yang. 2012. Processing a large number of continuous preference top-k queries. In *Proceedings of ACM Management of Data (SIGMOD)*. 397–408.