FIRAS: A Framework for Interval Range Search and Sampling

Daichi Amagata[*]
The University of Osaka
Suita, Osaka, Japan
amagata.daichi@ist.osaka-u.ac.jp

Panagiotis Simatis[*]
University of Ioannina
Ioannina, Greece
p.simatis@uoi.gr

Panagiotis Bouros
Johannes Gutenberg University Mainz
Mainz, Germany
bouros@uni-mainz.de

Nikos Mamoulis
Univ. of Ioannina & Archimedes, Athena RC
Ioannina, Greece
nikos@cs.uoi.gr

Abstract

Intervals are ubiquitous in many applications, including temporal and uncertain databases. Range search, which retrieves all intervals that overlap a given query interval, is a key operation in such applications. As data sizes grow, range search results can become large, overwhelming users and resulting in long search times. Obtaining random samples from a large search result is a promising approach that alleviates the above issues. While for some applications, sampling range query results is adequate, others may require the complete query result. Hence, a challenging question arises: can we design a framework that efficiently handles both range search and range sampling? This work provides a positive answer. We propose FIRAS, a framework that supports range search and sampling in $O(\log n + k)$ time and $O(\log^2 n + s)$ time, respectively, with $O(n)$ space, where k (s) is the range search result (sample) size, and n is the data size. FIRAS can also be used to know the result size k of a range query in $O(\log^2 n)$ time; subsequently, the issuer can decide whether to retrieve all results or random samples thereof in $O(k)$ or $O(s)$ time, respectively. Finally, we extend FIRAS to apply to evolving interval data, where queries interleave with updates and both have to be supported efficiently. Our extensive experiments on real-world datasets demonstrate the efficiency of FIRAS.

1 Introduction

Many applications, such as temporal databases [20, 45], uncertain databases [30, 63], spatial databases [41, 42], and financial databases [4], manage large sets of intervals. For example, in temporal and versioned databases, each record (version) is associated with a validity time interval. Uncertain databases [30] manage (i) values collected by sensors or (ii) anonymized values based on data generalization [56], where each uncertain value is represented as a range. Similarly, in cryptocurrency and stock market databases, where asset prices are recorded at regular time intervals (e.g., every second), each recorded price is represented as a [min, max] interval, representing the fluctuation between recordings.

Range query is the main search operation in the above applications. Given a set X of intervals and a query interval q, the range query retrieves $R \subseteq X$ such that each $x \in R$ overlaps q, i.e., there exists a shared value between q and x. In temporal databases, a range query retrieves records (versions) that were valid (or active) at some point within the query interval. This type of query is implemented in many relational databases, such as PostgreSQL[1] and Microsoft SQL Server[2]. In uncertain and financial databases, a range query finds data with intervals that include at least one value inside the query range [4, 25, 31]. Recently, [41, 42] developed a technique that approximates spatial polygons as lists of intervals of consecutive raster cells along a space-filling curve. Range searches on such interval lists can identify polygon intersections and topological relations between spatial objects. Due to these important applications, the problem of range search on interval data has been actively studied recently [18, 27, 31, 32, 47].

As interval data collections are becoming larger, the size of the query result, denoted by k ($k = |R|$), can also become large and may overwhelm the user. In addition, when k is large, range search is slow because it needs $\Omega(k)$ time. *Independent range sampling* (IRS) [1, 2, 4, 11, 13–15, 44, 54, 57, 61, 62] alleviates these problems, by allowing the user to retrieve s ($\ll k$) random samples from R. IRS guarantees that each sample is picked with probability $1/k$, which means that the samples are independent of any previous samples and queries. Thanks to this property, IRS is particularly useful for applications that want to obtain statistical information from R. For estimation tasks (e.g., approximate range aggregation queries), random samples of R provide an error-bounded estimation (from the Chernoff bound). Another application of IRS is visualizing the distribution of R. When k is large, visualization incurs a significant delay [61], whereas random samples of R alleviate this issue without losing the distribution [4]. Furthermore, IRS can be used as a (supporting) tool for query result diversification by proportional representation [46]. There are many diversity criteria [64], so it is difficult for ordinary users to select an appropriate one. When random samples are distributed evenly in the result space, they are regarded as diverse results [57], as it (approximately) covers the data space [7]. IRS can provide different samples for each execution, and these samples exhibit their distribution, which would suggest a useful diversification function, e.g., the Max-Min function [3, 39].

Motivation. Our work is motivated by the following limitations of previous work.

(1) *Existing access methods for intervals do not support efficient range search and IRS simultaneously.* Range search and IRS on interval data find many practical applications. Some applications require the complete range search result R, whereas for some others, obtaining

[*] Both authors contributed equally to this research and are sorted in alphabetical order. Daichi Amagata is the corresponding author and also belongs to Nagoya University.

[1] https://wiki.postgresql.org/wiki/Temporal_Extensions

[2] https://learn.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables?view=sql-server-ver17

random samples of R is more appropriate. In particular, when $|R| = k$ is large, applications suffer from the $\Omega(k)$ time and want to switch to IRS. For this, we should know k fast, without applying range search. To fully cover these requirements, we need a framework that can handle range search, range counting, and IRS using the same data structure(s). State-of-the-art access methods for intervals [18, 31, 32, 47] address only range queries but not IRS. They can only be used for IRS by first computing the entire R and then picking s random samples from R. Thus, they suffer from the $\Omega(k)$ cost. The time complexity of IRS should be $o(n) + O(s)$ [1, 2, 4, 44], where $o(n) = O(n^{1-\epsilon})$ and $0 < \epsilon < 1$.

(2) *State-of-the-art range search algorithms lack interesting time bounds.* Like IRS, the range search time should also be sensitive only to k. Although the state-of-the-art access methods for intervals [18, 31, 32, 47] perform well in practice, they are heuristically optimized for range search, and no output-sensitive bounds for these data structures are known. The interval tree [35] can achieve output-sensitive range search using $O(n)$ space, but it does not provide an optimal IRS algorithm [4].

(3) *The state-of-the-art IRS algorithm is not scalable.* An $O(\log^2 n + s)$ time IRS algorithm for interval data was presented in [4]. The time requirement for IRS is satisfied, but AIT, its data structure, consumes $O(n \log n)$ space. To scale well to large datasets, IRS (and range search) structures should have $O(n)$ space.

(4) *Non-trivial extension for evolving interval data.* Since interval data are often dynamic and associated with evolving domains (e.g., in temporal databases), they have to deal with new and evolving intervals efficiently. LIT [33] considers this setting and extends HINT [31, 32], a state-of-the-art range search structure, but it provides no search and update time bounds. Although AIT [4] can be extended for evolving interval data, it requires rebuilding to keep its $O(\log^2 n + s)$ IRS time. Because building an AIT takes $O(n \log n)$ time, it is not appropriate for evolving interval data.

Contribution. We address the above limitations of existing techniques and make the following contributions.

- *A novel framework for efficient range search/counting and IRS.* We propose *FIRAS* (Framework for interval range search/sampling), which enables theoretically and practically efficient range search, range counting, and IRS seamlessly. FIRAS manages X using two data structures, a sorted array and any interval data structure that supports stabbing queries. We apply a query result decomposition idea, which *simplifies* range search and IRS, and efficiently identifies the data space having only R. We prove that, with $O(n)$ space, FIRAS can solve range search and IRS in $O(\log n + k)$ and $O(\log^2 n + s)$ times, respectively. In addition, given a query interval, FIRAS can obtain its range search result size k in $O(\log^2 n)$ time; based on k, the user can then decide whether to retrieve the complete result R in $O(k)$ time or its random samples in $O(s)$ time.

- *Evolving domain interval tree for streaming interval data.* We extend FIRAS to handle *evolving* interval data, where new intervals can start or existing ones can end at the end of the (evolving) domain. We incorporate LIT's idea for managing intervals with only left-endpoint, represented as $[x.l, \infty)$ (infinite interval), and closed intervals, i.e., $[x.l, x.r]$, where $x.l$ and $x.r$ are respectively the left- and right-endpoints of interval x. Specifically, we adapt LIT's hashmap-based data structure for infinite intervals, while newly yielding bounded update and search times. For closed intervals, FIRAS for static interval data is employed. However, existing interval data structures are constructed for a static interval set. A straightforward solution for updating the interval data structure inside FIRAS is to rebuild it (e.g., with $O(n \log n)$ time for an interval tree) whenever a new interval appears.

Instead, we propose *evolving domain interval tree* (EDIT), which appends nodes and sub-trees to the existing structure, so that (i) the tree does not have to be reconstructed, (ii) existing intervals do not have to move between nodes, (iii) a new interval insertion is done in only $O(\log n)$ time on average, and (iv) the worst and amortized update times of EDIT are respectively $O(D)$ and $O(1)$ times, where D is the (evolving) domain size. In practice, since each interval endpoint defines a value in the domain, $D = O(n)$, clarifying the advantage of EDIT against the rebuilding-based approach.

Table 1 summarizes our theoretical results (shaded) and compares them with the existing state-of-the-art methods with known bounded time/space complexity.

Table 1: Comparison of time and space complexities. For static (dynamic) data, FIRAS uses an interval tree (EDIT) as its interval data structure.

Method	Range search time	IRS time	Space
Interval tree [35]	$O(\log n + k)$	$O(n)$	$O(n)$
AIT [4]	$O(\log n + k)$	$O(\log^2 n + s)$	$O(n \log n)$
FIRAS (static)	$O(\log n + k)$	$O(\log^2 n + s)$	$O(n)$
FIRAS (dynamic)	$O(\sqrt{n} + k)$	$O(\sqrt{n} + s)$	$O(n)$

- *Experiments on real-world datasets.* We conduct extensive experiments on real-world interval datasets with diverse distributions. In the static case, the experimental results demonstrate that FIRAS provides competitive performance for range and IRS queries with the state-of-the-art algorithms optimized for either range or IRS queries. FIRAS is the only method that can efficiently process both types of queries, not only theoretically but also practically, using only $O(n)$ space. The dynamic case observes similar results: FIRAS is update-friendly, and EDIT provides faster processing times for range queries than the extended HINT employed in LIT [33], the state-of-the-art range query processing algorithm for dynamic interval data. Furthermore, the performance of FIRAS in IRS is similar to that of an extended version of AIT (EDAIT), but FIRAS has a much lower update cost than EDAIT, while using only $O(n)$ space, as opposed to the $O(n \log n)$ space consumed by EDAIT.

2 Preliminaries

2.1 Problem Definition

Let X be a set of n intervals. We consider both the case where X is static and the case where X is evolving by a stream of time-ordered updates. We define each of these cases.

Definition 2.1 (Static interval set). When X is a static interval set, each interval $x \in X$ is represented as $x = [x.l, x.r]$, i.e., it is a pair of left-endpoint $x.l$ and right-endpoint $x.r$, where $x.l < x.r$.

Definition 2.2 (Evolving interval set). In this setting, the left- and right-endpoints of intervals arrive in time order from a stream of updates. Without loss of generality, we assume a discrete time domain $[1, D]$ (e.g., the UNIX time), where D increases over time. X contains all intervals that have appeared so far, and it may include intervals x that have started at time $x.l$ but not ended yet. Such intervals are represented as $x = [x.l, \infty)$, modeling the time span of a current version of a fact or an event, which started at time $x.l$ and is still valid. If x has its right-endpoint, it is represented in the same way as in the static case.

Evolving intervals are record versions in transaction-time temporal and multi-version databases [17, 33], where X stores validity periods of objects to support temporal range queries.

We say that x overlaps x' when

$$(x.l \leq x'.r) \wedge (x'.l \leq x.r). \tag{1}$$

The problems addressed in this paper are formally defined below.

Definition 2.3 (Range Search). Given a set X of intervals and a query interval $q = [q.l, q.r]$, this problem returns R, the set of intervals $\in X$ overlapping q. In the special case, where $q.l = q.r$, i.e., q is a single value, the problem is called stabbing query, and it retrieves every $x \in X$ such that $x.l \leq q \leq x.r$.

Definition 2.4 (Range Counting). Given a set X of intervals and a query interval $q = [q.l, q.r]$, this problem returns $k = |R|$, where R is defined in Definition 2.3.

Definition 2.5 (Independent Range Sampling). Given a set X of intervals, a query interval q, and a sample size s, this problem returns s independent random samples from R, each of which is picked with probability $1/k$.

In this paper, we study IRS with replacement. Lemma 3 of [44] proves that taking $2s$ ($s \leq k/3e$) random samples with replacement from R yields at least s random samples without replacement with probability at least $1/2$, where e is the natural number. Therefore, in the without replacement case, the theoretical time complexity results in this paper still hold by expectation.

Like most recent works on interval data management [4, 5, 8–10, 23, 25, 27, 28, 31–33], we assume that X is memory resident. Section 3 assumes that X is static (Definition 2.1), and Section 4 considers streaming intervals in an evolving domain (Definition 2.2). In the second case, interval updates are interleaved with queries.

2.2 Interval Tree

We review the interval tree (IT) [35, 38], the worst-case optimal data structure for interval data.

2.2.1 Construction and content. IT is a binary search tree which organizes a set X of n intervals to support stabbing and range queries. Given X, we compute the median value of the $2n$ endpoints in X, denoted by c_{root}. We partition X into three disjoint sets, $X_{root} = \{x \mid x \in X, x.l \leq c_{root} \leq x.r\}$, $X_l = \{x \mid x \in X, x.r < c_{root}\}$, and $X_r = \{x \mid x \in X, c_{root} < x.l\}$. The root node of the IT, u_{root}, maintains X_{root}, and the intervals in X_{root} are placed in two lists L^l_{root} and L^r_{root}. The intervals in L^l_{root} (L^r_{root}) are sorted in ascending (descending) order of left-endpoint (right-endpoint). X_l, and X_r (if non-empty) are respectively assigned to the left and right sub-trees of u_{root}, which are constructed recursively. Figure 1 illustrates an example of an IT for the thirteen intervals x_1 to x_{13} (shown at the bottom). The IT is built in $O(n \log n)$ time, and its space complexity is $O(n)$ [35]. Furthermore, since each node u of the IT is guaranteed to store at least one interval and each of its two children cannot contain in their sub-trees more than half of the intervals in the sub-tree rooted at u, the height of IT is $O(\log n)$.

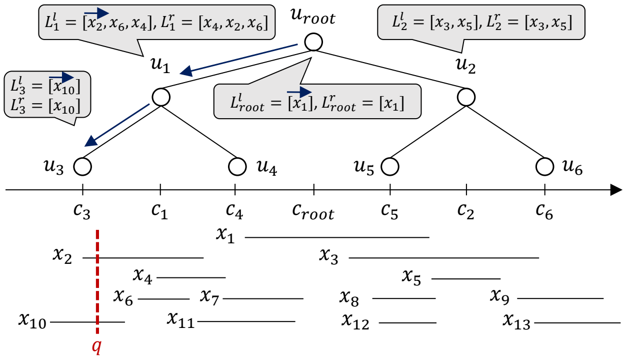
$L^l_1 = [\overrightarrow{x_2}, x_6, x_4], L^r_1 = [x_4, x_2, x_6]$
u_{root}
$L^l_2 = [x_3, x_5], L^r_2 = [x_3, x_5]$
u_1
u_2
$L^l_3 = [\overrightarrow{x_{10}}]$
$L^r_3 = [x_{10}]$
$L^l_{root} = [\overrightarrow{x_1}], L^r_{root} = [x_1]$
u_3
u_4
u_5
u_6
c_3
c_1
c_4
c_{root}
c_5
c_2
c_6
x_1
x_2
x_3
x_4
x_5
x_6
x_7
x_8
x_9
x_{10}
x_{11}
x_{12}
x_{13}
q

Figure 1: Example of an interval tree and a stabbing query q. Because $q < c_{root}$, we access L^l_{root}, find that no interval is stabbed by q, and traverse its left child node u_1. Because $q < c_1$, we sequentially access the intervals $\in L^l_1$. We terminate the scan when x_6 is accessed because $q < x_6.l$. Next, we traverse its left child node u_3 and repeat the same sequential scan. Because u_3 is a leaf node, we return $R = \{x_2, x_{10}\}$.

2.2.2 Stabbing query. Given a stabbing query q, we traverse the IT from u_{root}. If $q < c_{root}$ ($c_{root} < q$), we use L^l_{root} (L^r_{root}) and sequentially access the intervals in the list until we have $q < x.l$ ($q > x.r$); we then visit its left (right) child node and repeat the above until we have no more nodes to visit. Figure 1 showcases an example of stabbing query processing on the IT.

Lemma 2.6. *The IT can process a stabbing query in $O(\log n + k)$ time [35].*

2.2.3 Range query. Existing literature does not show how to process a range query in $O(\log n + k)$ time using the IT; we hereby prove:

Lemma 2.7. *The IT can process a range query in $O(\log n + k)$ time.*

Proof. For a query interval $q = [q.l, q.r]$, we traverse the IT from the root node u_{root}. If $q.r < c_{root}$ ($c_{root} < q.l$), we perform the same list scan as in stabbing query processing and visit u_{root}'s left (right) child node, as all intervals existing in the sub-tree rooted at its right (left) sub-tree cannot overlap with q. If $q.l \leq c_{root} \leq q.r$, we report all intervals in u_{root} as results and visit both its left and right child nodes. For each visited node, the search is applied recursively.

The above algorithm accesses at most two paths P_{left} and P_{right}, where, for each node u along P_{left} (P_{right}), $q.r < c$ ($c < q.l$). In each such node, the number of accessed intervals that do not contribute to the search result R is at most one. In addition, for each node u such that $q.l \leq c_u \leq q.r$, all intervals in u are query results; since each IT node contains at least one interval (by definition), the number of accessed intervals is $O(\log n + k)$ and the number of accessed nodes is also $O(\log n + k)$. Hence, the total time of the range query is $O(\log n + k)$. □

2.2.4 IRS query. To guarantee $1/k$ sampling probability for each $x \in R$ and $O(1)$ sampling time, the proof of Lemma 2.7 suggests that the IT needs to enumerate all nodes containing the intervals in R. This requires $\Omega(k)$ time, failing to obtain $o(n) + O(s)$ time.

The augmented interval tree (AIT) [4] avoids this issue. The main difference from IT is that AIT has *augmented lists* in its nodes with all intervals in their sub-trees. AIT needs to access at most $\log n$ nodes to cover all intervals in R [4]. Therefore, for every such node, AIT runs a binary search on the corresponding (augmented) list to find the index range that represents a set of intervals overlapping q in the node (or list). Since each index range size can be different, AIT builds an alias structure [59] to enable $O(1)$ time weighted random sampling[3]. AIT picks a random node from the alias structure, then samples a random interval falling in the index range from the corresponding list. This is repeated s times. Based on the augmented lists, AIT runs an IRS query in $O(\log^2 n + s)$ time while guaranteeing the $1/k$ sampling probability for each $x \in R$ [4]. However, AIT requires $O(n \log n)$ space because of the augmented lists.

3 FIRAS for Static Interval Data

This section presents FIRAS, a new framework that efficiently supports both range search/counting and IRS on interval data.

Main idea. The rationale behind FIRAS stems from the following observation.

Proposition 3.1. *Consider a range query $q = [q.l, q.r]$ on an interval collection X. The query result R can be decomposed into Q_1 and Q_2, where*

$$Q_1 = \{x \mid x \in X, q.l \leq x.r < q.r\}, \tag{2}$$

$$Q_2 = \{x \mid x \in X, x.l \leq q.r \leq x.r\}, \tag{3}$$

and $Q_1 \cap Q_2 = \emptyset$.

Proof. $Q_1 \cap Q_2 = \emptyset$ holds, since, for $x \in Q_1$ and $x' \in Q_2$, we have $x.r < q.r$ and $q.r \leq x'.r$. We therefore prove that $R = Q_1 \cup Q_2$. First, we focus on each interval $x \in X_1$, where $X_1 = \{x \mid x \in X, x.r < q.r\}$. If x overlaps q, x must have that $q.l \leq x.r$ from Equation (1), deriving Equation (2). We next focus on each interval $x' \in X_2$, where $X_2 = \{x' \mid x' \in X, q.r \leq x'.r\}$. If x' overlaps q, x' must have that $x'.l \leq q.r$ from Equation (1), which derives Equation (3). Because $X_1 \cup X_2 = X$, it is true that $R = Q_1 \cup Q_2$. □

Figure 2 illustrates an example of Proposition 3.1. The main merit of this decomposition is to replace a range query with two queries of equal or lower complexity, particularly for the IRS problem, as we show later. Specifically, the range search problem can be reduced to two easier problems: a one-dimensional (1D) range search and a stabbing query over intervals. (This also holds for the IRS problem.)

- Consider the set of all the right-endpoints of all intervals in X. Equation (2) suggests that a 1D range query on this set provides Q_1.
- Similarly, Equation (3) and Definition 2.3 suggest that a stabbing query of $q.r$ obtains Q_2.

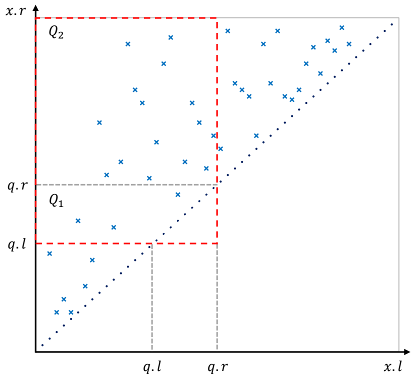

Figure 2: Illustration of Proposition 3.1. Each interval x is transformed to a 2D point (represented as ×) whose coordinates are $(x.l, x.r)$. A given query interval q is transformed into a two-sided orthogonal range query $(-\infty, q.r] \times [q.l, \infty)$ shown as a dashed rectangle. The query rectangle can be split into a 1D range query in the $x.r$ domain (Q_1) and a stabbing query (Q_2) on the intervals.

Thus, by preparing data structures for 1D range searches and stabbing queries over intervals, we can efficiently solve the range search and IRS problems. We exploit this new idea to achieve theoretically and practically efficient range search/counting and IRS.

Remark. Given two intervals $x = [x.l, x.r]$ and $y = [y.l, y.r]$, Dignös et al. [37] showed that x overlaps y iff $x.l \leq y.l \leq x.r$ or $y.l < x.l \leq y.r$. Using this, they convert the problem of overlap interval join between two interval collections into two *range joins*, where all intervals in one collection are used as range queries on the left endpoints of the other collection. The *range query result decomposition* suggested in Proposition 3.1 is similar to the *overlap predicate decomposition* in [37]; however, *the motivation and application of these two decompositions differ.* Unlike showing that the interval join problem can be decomposed into two range joins, Proposition 3.1 shows, for the first time (to the best of our knowledge), that a range search on interval data can be decomposed into (1) a stabbing query on interval data and (2) a range search on 1D points. Our finding cannot be derived from [37] and leads to a different query processing strategy from [37]. In Section 6, we further discuss this point.

3.1 Data Structures

To take advantage of Equations (2) and (3), FIRAS manages X with two data structures. To compute Q_1 efficiently, FIRAS stores the right-endpoints of all intervals in X in a sorted array $\mathcal{A}$. In addition, FIRAS manages X in an interval data structure $\mathcal{I}$ that supports stabbing queries, for the computation of Q_2. In this section, we use an interval tree as the default $\mathcal{I}$ due to its optimal space and time complexity. The two data structures ($\mathcal{A}$ and $\mathcal{I}$) of FIRAS are built in pre-processing and used for both range and IRS queries.

[3] For n objects each of which is associated with a weight w, Walker's alias method can sample an object of weight w_i with probability $\frac{w_i}{\sum_n w}$ in $O(1)$ time. Its structure, called alias, is built in $O(n)$ time [59].

The sorted array $\mathcal{A}$ can be created in $O(n \log n)$ time, and $\mathcal{I}$ can also be constructed in $O(n \log n)$ time (see Section 2.2). Each data structure needs $O(n)$ space. Hence:

Lemma 3.2. *FIRAS needs $O(n \log n)$ pre-processing time and $O(n)$ space.*

3.2 Independent Range Sampling

For ease of presentation, we first discuss how to process IRS queries with FIRAS, which is described in Algorithm 1. This section proves that FIRAS can yield s random samples of R in $O(\log n + s)$ time with $O(n)$ space, i.e., a better result than those of the interval tree and AIT.

Given a query interval q and a sample size s,

(1) FIRAS first runs two binary searches on $\mathcal{A}$: one using $q.l$ as query point and another using $q.r$. These binary searches provide the indices of $\mathcal{A}$, a, and b, respectively. Notice that $[a, b)$ (or $[a, b-1]$) represents the index range, where every $\mathcal{A}[j]$ such that $j \in [a, b)$ is contained in q, see Equation (2). Also, $b - a = |Q_1|$.

(2) Next, FIRAS runs a variant of stabbing query on $\mathcal{I}$. The difference to the stabbing query processing introduced in Section 2.2 is as follows. For each accessed node u_i along the search path, we run a binary search on L_i^l or L_i^r (based on the condition introduced in Section 2.2) to find the index range $[a_i, b_i]$ of intervals overlapping q, as with the above step. We use Walker's alias structure $\mathcal{AS}$ of all accessed nodes with their ranges (i.e., $[a_i, b_i]$) to enable $O(1)$ time weighted random sampling [59]. At the same time, we obtain $|Q_2|$ by summing the lengths of the ranges.

(3) After that, FIRAS picks s random samples as follows. FIRAS generates a random value v in $[1, |Q_1| + |Q_2|]$. If $v \le |Q_1|$, FIRAS randomly samples an index, idx, from $[a, b)$ and returns the interval corresponding to $\mathcal{A}[\text{idx}]$. Otherwise, FIRAS samples a random node u_i in $\mathcal{AS}$, samples a random index, idx, from $[a_i, b_i]$, and returns the interval $L_i^l[\text{idx}]$ or $L_i^r[\text{idx}]$.

3.2.1 *Analysis.*

We prove one of our main results:

Theorem 3.3. *With $O(n)$ space, FIRAS can process an IRS query correctly in $O(\log n + s)$ and $O(\log^2 n + s)$ times in the average and worst cases, respectively.*

Proof. We first consider the time and space complexities.

Step (1): Two binary searches on $\mathcal{A}$ need $O(\log n)$ time.

Step (2): FIRAS runs the stabbing query variant. We prove that this time is $O(\log n)$ in the average case. Since $\mathcal{I}$ has $O(n)$ nodes, each node manages $n/O(n) = O(1)$ intervals in the average case. Hence, the cost of a binary search on each accessed node is $O(1)$. Lemma 2.6 guarantees that FIRAS accesses at most $\log n$ nodes. Based on the above, the average case needs $O(\log n)$ time. The worst case appears when FIRAS accesses at most $\log n$ nodes with $O(n)$-sized lists. In this case, the cost of a binary search on each accessed node is $O(\log n)$, and the time of the stabbing query variant is $O(\log^2 n)$. The alias structure can be built in $O(\log n)$ time, since FIRAS accesses at most $\log n$ nodes.

Step (3): If $v \le |Q_1|$, it is trivial that a random interval x is obtained in $O(1)$ time. Otherwise, we pick an accessed node of $\mathcal{I}$ randomly, and $\mathcal{AS}$ achieves this weighted random sampling in $O(1)$ time [59]. Obtaining a random index from the corresponding range needs $O(1)$ time. Picking s samples therefore needs $O(s)$ time.

Algorithm 1: IRS on FIRAS with interval tree

```
Input: FIRAS of $X$, $q = [q.l, q.r]$, $s$
$S \leftarrow \emptyset$
$a \leftarrow$ Binary-Search($\mathcal{A}, q.l$), $b \leftarrow$ Binary-Search($\mathcal{A}, q.r$)
$|Q_i| \leftarrow b - a$, $\mathcal{R} \leftarrow \emptyset$, IRS-Stabbing-Query($u_{root}, q.r, \mathcal{R}$)
$|Q_2|, \mathcal{AS} \leftarrow$ Alias-Build($\mathcal{R}$)
while $|S| < s$ do
    $v \leftarrow$ a random real value $\in [1, |Q_1| + |Q_2|]$
    if $v \le |Q_1|$ then
        idx $\leftarrow$ a random integer in $[a, b-1]$, $S \leftarrow S \cup \{\mathcal{A}[\text{idx}]\}$
    else
        $u_i \leftarrow$ a random node in $\mathcal{AS}$
        idx $\leftarrow$ a random integer in $[a_i, b_i]$
        if $[a_i, b_i]$ is obtained from $L_i^l$ then $S \leftarrow S \cup \{L_i^l[\text{idx}]\}$
        else $S \leftarrow S \cup \{L_i^r[\text{idx}]\}$
return $S$
Function IRS-Stabbing-Query($u_i, q.r, \mathcal{R}$):
    if $q.r < c_i$ then
        $j \leftarrow$ Binary-Search($L_i^l, q.r$)
        if $j \ge 1$ then $\mathcal{R} \leftarrow \mathcal{R} \cup \langle u_i, [1, j] \rangle$
        if $u_i^l$ (the left child node of $u_i$) exists then
            IRS-Stabbing-Query($u_i^l, q.r, \mathcal{R}$)
    else if $c_i < q.r$ then
        $j \leftarrow$ Binary-Search($L_i^r, q.r$)
        if $j \ge 1$ then $\mathcal{R} \leftarrow \mathcal{R} \cup \langle u_i, [1, j] \rangle$
        if $u_i^r$ (the right child node of $u_i$) exists then
            IRS-Stabbing-Query($u_i^r, q.r, \mathcal{R}$)
    else
        $\mathcal{R} \leftarrow \mathcal{R} \cup \langle u_i, (1, |L_i^l|) \rangle$
```

From the above analysis, the average and worst times are respectively $O(\log n + s)$ and $O(\log^2 n + s)$. Additionally, the alias structure requires $O(\log n)$ space, as it needs $O(m)$ space for m components. Combining this fact and Lemma 3.2, the $O(n)$ space claim holds.

Next, we prove that each sample is picked uniformly at random, i.e., with probability $1/k$. In the case where an interval is sampled from $\mathcal{A}$, its sampling probability is $|Q_1|/(|Q_1| + |Q_2|) \times 1/|Q_1| = 1/(|Q_1| + |Q_2|) = 1/k$. In the case where an interval is sampled from $\mathcal{I}$, we assume that the index of this interval is in the range $[a_i, b_i]$ of the corresponding sampled node. Then, its sampling probability is $|Q_2|/(|Q_1|+|Q_2|) \times (b_i - a_i + 1)/|Q_2| \times 1/(b_i - a_i + 1) = 1/(|Q_1| + |Q_2|) = 1/k$. □

The IRS problem requires $\Omega(s)$ time, so Theorem 3.3 shows that our result is optimal up to the (poly)logarithmic factor.

3.2.2 *Comparison with state-of-the-art.*

The current state-of-the-art method for IRS over intervals is due to [4], and its technique requires $O(\log^2 n + s)$ time using $O(n \log n)$ space. Theorem 3.3 shows that our result is theoretically better, as we obtain the same theoretical worst time while reducing the space complexity. Also, Section 2.2 shows that using only the interval tree cannot guarantee efficient IRS. These facts highlight the contribution of FIRAS.

3.2.3 Range counting. Theorem 3.3 straightforwardly derives that FIRAS efficiently processes a range counting.

Corollary 3.4. *FIRAS can process a range counting query correctly in $O(\log^2 n)$ time with $O(n)$ space.*

3.3 Range Search

FIRAS takes a similar approach to its IRS query processing algorithm. Given a query interval q,

(1) FIRAS runs two binary search on $\mathcal{A}$, as with the IRS case. Then, FIRAS scans $\mathcal{A}$, from the index found by the binary search of $q.l$ to the last index found by the binary search of $q.r$. During the scan, it outputs the intervals corresponding to the accessed elements without comparisons.
(2) Next, FIRAS runs a stabbing query on $\mathcal{I}$.

3.3.1 Analysis. Lemmas 2.6 and 3.2 and Theorem 3.3 derive:

Theorem 3.5. *FIRAS can correctly process a range query in $O(\log n + k)$ time with $O(n)$ space.*

3.3.2 Comparison with state-of-the-art. Theorem 3.5 shows that FIRAS guarantees the state-of-the-art result theoretically. Its optimality is guaranteed from the same rationale as the IRS case. Although Theorem 3.5 shows that FIRAS has the same theoretical result as the interval tree, FIRAS has a better practical efficiency due to the following two observations. (i) Recall that FIRAS employs a sorted array: its scan cost is much cheaper than the cost for traversing all nodes that contain the range search result. (ii) The *practical* stabbing query cost is guaranteed to be cheaper than the range query cost.

3.4 Seamless Property

We now clarify how to efficiently choose between range search and sampling. Recall Corollary 3.4: a user can know k in $O(\log^2 n)$ time. If k is large, the user can opt for IRS, and we proceed to step (3) of our IRS algorithm to obtain s random samples of R. On the other hand, if k is not too large, the user can select range search, and we enumerate all intervals inside the index ranges identified in the steps (1) and (2) of the IRS algorithm.

The state-of-the-art range search algorithms (e.g., the access methods on interval tree and HINT [31]) do not support efficient IRS, so they cannot efficiently support the above selection. Although AIT [4] can do range counting in $O(\log^2 n)$ time, it consumes $O(n \log n)$ space, which is much higher than the $O(n)$ space of FIRAS.

3.5 Generalization

3.5.1 IRS and range search algorithms. FIRAS accepts any interval data structure for the stabbing query processing in step (2) of the algorithms in Sections 3.2 and 3.3. For IRS, the objective in this step is to find (i) the nodes (or corresponding components) in the data structure, which contain the query results Q_2 and (ii) the index *ranges* of the data structure (e.g., list) of each node that has these results. FIRAS follows this approach for a given interval data structure. For range search, FIRAS simply enumerates the intervals inside the index ranges, as discussed in Section 3.4.

3.5.2 Analysis. We analyze the IRS and range search performances of FIRAS with an arbitrary interval data structure. The following results are derived from Theorems 3.3 and 3.5, respectively.

Corollary 3.6. *With $S(n) + O(n)$ space, FIRAS needs $O(\log n + s) + T_{vstab}(n)$ time to pick s intervals from R uniformly at random, where $S(n)$ is the space complexity of a given interval data structure and $T_{vstab}(n)$ is the processing time of the stabbing query variant of the given interval data structure.*

Corollary 3.7. *With $S(n) + O(n)$ space, FIRAS needs $O(\log n + |Q_1|) + T_{stab}(n)$ time to return R, where $T_{stab}(n)$ is the time of processing a stabbing query on the given interval data structure.*

For example, since $T_{vstab}(n)$ ($T_{stab}(n)$) of HINT [31] is $O(n)$, if FIRAS employs a HINT for stabbing queries, it requires $O(n)$ time to process an IRS (a range) query.

4 FIRAS for Evolving Interval Data

This section assumes the streaming setting introduced in Section 2.1. Since X evolves over time, the data structures employed in FIRAS have to be updated efficiently. As long as these data structures are up-to-date, Section 3 demonstrates that FIRAS processes range search/counting and IRS queries efficiently for intervals having both left- and right-endpoints. However, since Section 3 does not consider intervals without right-endpoints, FIRAS needs to be extended to handle them. The challenge in this section is to extend FIRAS so that it has low (and bounded) update and querying costs.

4.1 Data Structures

For evolving interval data, FIRAS manages infinite intervals (i.e., intervals with only left-endpoint) and closed intervals (i.e., intervals with both left- and right-endpoints) in different data structures. Each infinite interval $x = [x.l, \infty)$ can be regarded as a one-dimensional point, and x overlaps q if $x.l \leq q.r$.

As with the case of static interval data, for the closed intervals, FIRAS uses a sorted array $\mathcal{A}$ and an interval data structure $\mathcal{I}$. For the infinite intervals $[x.l, \infty)$, FIRAS adopts the corresponding data structure of LIT [33]; i.e., a sequence of *buffers* $B_1, ..., B_b$ that store the left-endpoints. Each interval belongs to one buffer and all $x.l$ in buffer B_i are smaller than all $x.l$ in buffer B_{i+1}. Figure 3 illustrates how to manage intervals in FIRAS for evolving X. To enable $o(n)$ time, the capacity of each buffer is limited to $\lceil \sqrt{n} \rceil$.

Each buffer is implemented as a gapless hashmap [51], which has $O(1)$ insertion/deletion time, while yielding a practically fast scan due to its contiguous storage. To achieve this contiguous storage, this hashmap is implemented based on an (unsorted) array. For each buffer B_i, we keep track of the minimum $x.l$ in B_i. Its merit is that we can know which buffer is fully or partially covered by $(-\infty, q.r]$. Consider the first i such that $q.r < \min_{B_i} x.l$, then B_{i-1} is regarded as the partially covered buffer, and $B_1, ..., B_{i-2}$ are fully covered buffers. ($B_i, ..., B_b$ can be ignored.) For fully covered buffers, a range query can output the intervals in these buffers without any comparisons. In addition to this practical efficiency, we can support theoretically efficient range and IRS queries on these buffers while keeping their update cost low. (LIT does not clarify how to yield interesting theoretical guarantees w.r.t. update and search performances.)

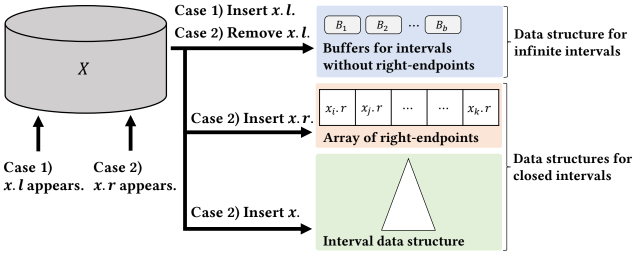

Figure 3: Example of FIRAS for evolving X and how to deal with endpoint insertions. A sequence of buffers is used to manage open-ended intervals, whereas closed intervals are maintained with a sorted array and an interval data structure similar to FIRAS for static interval data.

4.1.1 Overview of the data structure updates. When $x.l$, i.e., the left-endpoint of a new interval x, appears, FIRAS inserts this point into the last buffer. If this buffer already contains $\lceil\sqrt{n}\rceil$ points, we make a new buffer, $x.l$ is inserted into this new buffer, and it becomes its minimum $x.l$. When $x'.r$, i.e., the right-endpoint of x', appears, FIRAS removes $x'.l$ from the corresponding buffer, appends $x'.r$ to the sorted array $\mathcal{A}$, and inserts x' into $\mathcal{I}$. If the buffer, say B_i, becomes sparse and merging it with its adjacent buffer (B_{i-1} or B_{i+1}) yields a buffer with at most $\lceil\sqrt{n}\rceil$ points, we merge them.

The aforementioned update procedures yield the following update analysis. Let $U(n)$ be the cost of updating $\mathcal{I}$.

Lemma 4.1. *When the left-endpoint of a new interval appears, FIRAS needs $O(1)$ update time. When the right-endpoint of an existing interval x appears, FIRAS needs $O(\sqrt{n}) + U(n)$ update time.*

Proof. When a new $x.l$ appears, FIRAS only needs to insert this value into the last or new buffer, which requires $O(1)$ time. When the right-endpoint $x.r$ of an existing interval x appears, FIRAS first removes the left-endpoint of x from the corresponding buffer. Finding this buffer needs $O(\log n)$ time, by binary-searching the array having the minimum left-endpoints of the buffers, and deleting this point from this buffer needs $O(1)$ time. If merging two buffers occurs, this can be done in $O(\sqrt{n})$ time, by inserting all points in one of the buffers into the other, since buffers have $O(\sqrt{n})$ points. FIRAS next inserts the right-endpoint into $\mathcal{A}$, and this cost is $O(1)$ since the points in $\mathcal{A}$ are sorted based on the right-endpoints. Last, FIRAS inserts the new closed interval $[x.l, x.r]$ into $\mathcal{I}$, which costs $U(n)$ time from the definition. □

Skewed data. When numerous new intervals with the same $x.l$ arrive, the last buffer may overflow without the possibility of splitting it, as buffers have to span disjoint subdomains. Since $x.l$ is the search key, for multiple values having the same $x.l$, we can keep, in the buffer, a single $x.l$ value, and link it to a hashmap, where we manage the interval identifiers having $x.l$. Note that if $x.l$ is a search result, then all associated intervals are results. Intervals in the hashmap linked to $x.l$ are indexed by their identifiers, which are used as a secondary key for insertions and deletions. This way, we can keep the $O(\sqrt{n})$ buffer capacity in terms of distinct $x.l$ values and the corresponding complexities. This is similar to the *heavy columns* idea applied in [28, 29].

4.1.2 Comparison with a balanced search tree. As shown in [57], the worst-case optimal updatable 1D IRS index is a balanced search tree (i.e., an augmented variant of B-tree or red-black tree). Despite its theoretical optimality, such a structure is slower in practice than a sequence of hashmaps, in terms of updates in the streaming setting [33]. In addition, the 1-sided range queries $x.l \leq q.r$ that we have to support are costly on (balanced) trees [33]. To achieve good practical performance, as FIRAS has to support both range and IRS queries efficiently, it employs a sequence of hashmaps rather than a search tree tailored for IRS.

4.1.3 Update time analysis of the state-of-the-art interval data structures. The interval tree does not support incremental updates (insertions), meaning that it theoretically needs *re-building* to maintain a logarithmic height.[4] Therefore, $U(n)$ of the interval tree is $O(n \log n)$. However, the logarithmic method [19] can amortize the update cost. Let $\mathcal{S}$ denote a given interval data structure. The idea of the logarithmic method is to manage $h = O(\log n)$ interval data structures, $\mathcal{S}_0$, $\mathcal{S}_1$, ..., $\mathcal{S}_{h-1}$, such that $\mathcal{S}_j$ maintains 2^j intervals. (Each interval belongs to only a single interval data structure.) Given a new interval x, the logarithmic method identifies the smallest j such that $\mathcal{S}_j$ has no intervals. It then builds $\mathcal{S}_j$ based on x and all intervals in $\mathcal{S}_0$, $\mathcal{S}_1$, ..., $\mathcal{S}_{j-1}$ (these interval data structures are destroyed). Building $\mathcal{S}_j$ needs $U(2^j)$ time, and its amortized time is $U(2^j)/2^j$. Since each interval is moved at most $O(h)$ times for n insertions, the amortized insertion time per interval is $\frac{U(2^j)\log n}{2^j}$.

For example, as for the interval tree, since $U(2^j) = O(2^j \log 2^j)$ and $j = O(\log n)$, it incurs $O(\log^2 n)$ amortized update time. (AIT also has this amortized time because its building algorithm essentially follows that of the interval tree.) Although the logarithmic method can keep a small amortized update time, it cannot avoid the large *worst* update time, and it always needs to rebuild one interval data structure, say $\mathcal{S}_j$.

4.2 EDIT: Evolving Domain Interval Tree

As discussed in Section 4.1, using an interval tree as the interval data structure $\mathcal{I}$ in FIRAS incurs a high update cost in the worst case. To overcome this issue, we propose to use, as $\mathcal{I}$, a new variant of the interval tree, the *evolving-domain interval tree* (EDIT).

Each node of the EDIT has the same components as those of an interval tree; however, unlike the interval tree, the *EDIT grows incrementally*. The nodes in the EDIT are created when the domain $[1, D]$ grows. Specifically, x, a new closed interval inserted to the EDIT, always ends at the end of the current domain (i.e., $x.r = D$) and may trigger the growth of the EDIT. For ease of presentation, we assume discrete time semantics as in temporal databases [17], i.e., D is incremented by one for each domain update. Let D_{edit} be the domain managed by the EDIT; $D_{edit} = 0$ at initialization. As explained below, both $D_{edit} > D$ and $D_{edit} \leq D$ are possible.

4.2.1 Updating EDIT. We add new nodes into the EDIT only when D exceeds D_{edit} and D is an odd integer.[5]. (For example, when $D \leq 2$, the EDIT has only one node with $c = 1$.) In this case, we

[4] In [33], the HINT [31] data structure for closed intervals is extended for evolving time domains, but its update efficiency is *not* analyzed/known.

[5] This condition is based on the assumption that $x.l < x.r$. Dealing with the case of $x.l \leq x.r$ is straightforward, so it is omitted.

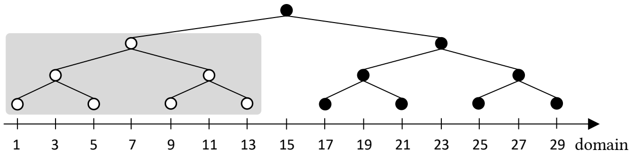

Figure 4: Example of adding nodes to the EDIT. The nodes in the shadow are existing ones, whereas black nodes are newly added when $D = 15$.

add new nodes to the EDIT, whose c are D, $D+2$, ..., and $2D-1$. Its structure follows a balanced binary search tree. Thus, the new node, such that $c = D$, becomes the new root node, and its right sub-tree is created accordingly. We then set $D_{edit} = 2D - 1$.

Example 4.2. Figure 4 illustrates an example of adding nodes to the EDIT. When $D = 14$, the EDIT consists of the seven nodes in the shadowed frame, and $D_{edit} = 13$. Now assume that $D = 15$. Since D is odd and $D > D_{edit}$, we add nodes corresponding to 15, 17, ..., 29 (= $2D - 1$) into the EDIT, which are represented by the black nodes. Then, we set $D_{edit} = 2D - 1 = 29$.

4.2.2 Interval insertion to EDIT. Recall that, when $x_j.r$ appears, FIRAS inserts interval x_j to the EDIT. Each interval insertion to the EDIT traverses the tree to find the first node u such that $x_j.l \leq c_u \leq x_j.r$. Given x_j at time D (i.e., $x_j.r = D$), we start traversal from the EDIT's root. If $x_j.r < c_{root}$ ($c_{root} < x_j.l$), we traverse its left (right) child node and repeat recursively. As soon as we find a node u such that $x_j.l \leq c_u \leq x_j.r$, we insert x_j into L^l and L^r and terminate.

4.2.3 Analysis. Compared to a data structure that needs to be rebuilt using the logarithmic method, EDIT has the following merits:

Lemma 4.3. *When D is odd and $D > D_{edit}$, inserting new nodes to the EDIT takes $O(D)$ time; the amortized insertion time is $O(1)$.*

Proof. When D is odd and $D > D_{edit}$, the number of nodes added to the EDIT is $O(D)$, which is straightforward from the fact that the new nodes have $c = D, D+2, ..., 2D-1$. Because each node has a fixed c, building the right sub-tree of the new root node needs $O(D)$ time. It is important to note that, when adding nodes to the EDIT, it does not need to move intervals in each node to the new nodes. When the node addition is triggered, all existing intervals x have $x.r < D$. Since all new nodes have $c \geq D$, the existing intervals do not have $x.l \leq c \leq x.r$ for these new nodes. Therefore, when adding nodes, the EDIT needs $O(D)$ update time. Since the EDIT has $O(D)$ nodes, for each increment of D, we need $O(D)/D = O(1)$ amortized time to update the EDIT. □

Lemma 4.4. *The insertion of x into the EDIT requires $O(\log D + n/D)$ time in the average case.*

Proof. The tree traversal way for the insertion of x is essentially the same as that for a stabbing query on the IT. Because the height of the EDIT is $O(\log D)$, finding the first node such that $x.l \leq c \leq x.r$ needs $O(\log D)$ time. Since each node has $O(n/D)$ intervals on average, inserting x into L^l and L^r needs $O(n/D)$ and $O(1)$ times, respectively. This is because of the following three observations. (i) Each node has $O(n/D)$ intervals on average. (ii) The intervals in L^l are sorted based on the left-endpoints, and insertions of the left-endpoints come in a random order. (iii) The intervals in L^r are sorted based on the right-endpoints, so x can be appended at the end of L^r. Thus, the total cost is $O(\log D + n/D + 1) = O(\log D + n/D)$. □

Real-world datasets usually have $D = O(n)$. In this case, each insertion needs $O(\log n)$ on average. This observation suggests that the EDIT has a superior update performance to the logarithmic method-based approach. Even in the worst case, the insertion of x into the EDIT requires $O(n)$ time, which is still better than the $O(n \log n)$ worst time of the logarithmic method. The space complexity of EDIT is comparable to that of the interval tree.

Corollary 4.5. *The space complexity of the EDIT is $O(\max\{n, D\})$.*

In the dynamic case, FIRAS employs EDIT as its default interval data structure.

4.3 Independent Range Sampling

The main difference from the static case is the need to sample infinite intervals from the buffers. Thanks to our framework, sampling infinite intervals is easy.

Specifically, given a query interval q and a sample size s,

(1) For each interval x in the buffers, if $x.l \leq q.r$, x overlaps q because $x.r > D$. Therefore, we access fully and partially covered buffers. Note that the number of partially covered buffers is at most one. For this buffer, we scan it to obtain the intervals overlapping q and keep them in an array S. For each fully covered buffer, we obtain the number of intervals in it. We build an alias structure for these buffers and S. Let cnt be the total number of intervals overlapping q in the buffers.
(2) FIRAS runs the same operations for the sorted array $\mathcal{A}$ and the EDIT as those for the static X case. Note that Q_1 and Q_2 here are defined from a set of intervals having both left- and right-endpoints.
(3) For random sampling, FIRAS generates a random value v in $[1, |R|]$. If $v \leq cnt$, FIRAS samples a random interval with the alias structure built in the first step. Otherwise, FIRAS runs the sampling algorithm in Algorithm 1.

We prove that FIRAS can achieve $o(n) + O(s)$ time even in the dynamic case.

Theorem 4.6. *For evolving X, FIRAS needs $O(\sqrt{n} + s)$ time to pick s intervals from R uniformly at random.*

Proof. The sampling (probability) correctness is trivial from Theorem 3.3, so we focus on the time complexity. The main difference from the proof of Theorem 3.3 is to analyze the IRS performance on the buffers and EDIT. The first step needs $O(\sqrt{n})$ time, because the number of buffers is $O(\sqrt{n}) = n/O(\sqrt{n})$, and each buffer contains at most $\lceil \sqrt{n} \rceil$ intervals by the capacity definition. Thus, building the alias structure also needs $O(\sqrt{n})$ time. Sampling an interval from this alias structure needs $O(1)$ time. Recall that $T_{vstab}(n)$ is the processing time of the stabbing query variant of a given interval data structure. As EDIT is based on the interval tree structure, the proof of Theorem 3.3 derives the following fact. For EDIT, $T_{vstab}(n) = O(\log D \log(n/D))$ in the average case, whereas $T_{vstab}(n) = O(\log D \log n))$ in the worst case. Summarizing the above results, FIRAS needs $O(\sqrt{n} + s)$ time. □

This result also proves that FIRAS can perform range counting in $O(\sqrt{n})$ time in the evolving X case.

4.4 Range Search

Given a query interval q, FIRAS runs the same algorithm as that in Section 3.3. Then, FIRAS runs a 1-dimensional range query of $(-\infty, q.r]$ on the buffers to find all intervals x such that $x.l \leq q.r$.

From Theorem 4.6, the following result is clear.

Corollary 4.7. *For evolving X, FIRAS needs $O(\sqrt{n} + k)$ time to return R.*

FIRAS achieves $o(n) + O(s)$ time for IRS and $o(n) + O(k)$ time for range search, even in the case of evolving intervals in an evolving domain, without incurring super-linear update cost.

4.5 General Update Support

Although we assume streaming intervals, FIRAS can also support a dynamic collection of closed intervals with arbitrary updates. When a new closed (and maybe out-of-order) interval x appears, it is directly inserted into EDIT, as shown in Section 4.2, which needs $O(\log D + n/D)$ average time (see Lemma 4.4). In addition, its right-endpoint $x.r$ must be inserted into the sorted array $\mathcal{A}$. To facilitate ad-hoc insertions in a logarithmic time, $\mathcal{A}$ should be replaced by a B-tree. When an existing interval x' is deleted, it is removed from the two lists in the corresponding node of the EDIT in $O(n/D)$ average time, which is straightforwardly derived from the proof of Lemma 4.4. Removing its right-endpoint $x'.r$ from $\mathcal{A}$ takes logarithmic time, if $\mathcal{A}$ is implemented as a B-tree. The variant of B-tree suggested in [44] can be used for the processing of one-dimensional IRS queries in $O(\log n + s)$ expected time.

5 Experimental Evaluation

This section reports our experimental results. The experiments were conducted on an Ubuntu 24.04.3 LTS machine equipped with an Intel Core i7-14700K 5.6GHz CPU and 64GB of RAM. We implemented all methods in C++, compiled using gcc (v13.3.0) with -O3, -mavx, and -march=native flags activated. Our code available at https://github.com/sigmod26firas/code.

5.1 Setup

We used five real-world datasets shown in Table 2. BOOKS [23] stores the periods of time when books were lent out by Aarhus city libraries in 2013. BTC [4] contains Bitcoin's historical price intervals. Their low and high prices were used as the left- and right-endpoints, respectively. RENFE [4] is Spanish rail trip data, and we used the departure time and arrival time as the left- and right-endpoints, respectively. TAXIS [23] contains the pick-up and drop-off times of taxi trips in New York City from 2009. WEBKIT [36] consists of the file history records in the git repository of the Webkit project from 2001 to 2016; the intervals indicate the periods during which a file did not change.

Regarding the static data case, we evaluated the performance of each method by executing 10,000 queries, uniformly positioned within the domain, while varying the extent of each query interval from 0.1% to 10%. For IRS, we additionally varied the requested sample size from 10 to 1,000. Last, for the evolving data case, we considered a setting similar to [33]. We created an event stream for each dataset by splitting its intervals into left-point and right-point arrival events, and interleaving 10,000 queries. A query was placed at the timestamp of a randomly chosen interval event, and its length (extent) was fixed to a percentage (0.01%, 0.1%, 1%, or 10%) of the domain extent.

Table 2: Dataset statistics

Dataset	BOOKS	BTC	RENFE	TAXIS	WEBKIT
n	2,050,707	2,538,921	38,753,060	106,685,540	2,347,346
Dom. size	31,413,600	6,876,400	52,163,400	79,901,357	461,829,284
Min len.	3,600	1	1,320	1	1
Avg. len.	5,771,887	2,291	9,120	663	33,206,291
Avg. len. (%)	18.4	0.033	0.017	0.002	7.22
Max len.	31,413,600	547,077	44,700	2,618,881	461,815,512

5.2 Static Data Case

We start off with the static interval data case. We evaluated the following methods in our experiments:

- IT (Interval Tree) [35]: For range queries, we used the algorithm described in Section 2.2. For IRS, IT first identifies all nodes containing R, then runs similar operations as AIT.
- HINT [27, 32]: This is the state-of-the-art interval data structure for range queries. The intervals are distributed in a hierarchy of 1D-grids, and intervals are replicated to the minimum number of partitions they overlap. HINT does not guarantee $o(n)$ time for range search and does not support IRS. Thus, for each IRS query, HINT first computes R and then randomly samples s intervals from it, resulting in $O(n)$ time.
- AIT [4]: This is the state-of-the-art interval data structure for IRS. Although AIT was devised for the IRS problem, it can answer range queries in $O(\log n + k)$ time like IT. AIT consumes $O(n \log n)$ space. Compared to [4], we improved AIT to define a single augmented list per tree node, reducing its space requirements by (approximately) half, without losing correctness.
- FIRAS: For its interval data structure $\mathcal{I}$, we used an IT. Hence, FIRAS has the theoretical results proved in Section 3. We do not use HINT as $\mathcal{I}$ in FIRAS, see Section 3.5. Using AIT as $\mathcal{I}$ in FIRAS makes no sense, since stabbing queries on AIT have the same cost as stabbing queries on IT, whereas AIT consumes more space than IT.

5.2.1 *Pre-processing Time and Memory Usage.*

Table 3 reports the pre-processing time of each method. IT achieves the fastest construction in most datasets, except TAXIS, where HINT is faster to build; TAXIS contains extremely short intervals, which are exclusively assigned to the bottom HINT levels, resulting in low replication and fast construction. FIRAS has a fixed construction overhead compared to IT, since it also has to build the auxiliary array $\mathcal{A}$. AIT is significantly slower across all datasets as expected, due to the costly creation of an augmented list at each node. Figure 5 plots the construction cost for different fractions of each dataset, showing that FIRAS is scalable (see Lemma 3.2).

Table 4 compares the memory consumption of each method. IT and FIRAS have the smallest footprints, as they both have $O(n)$

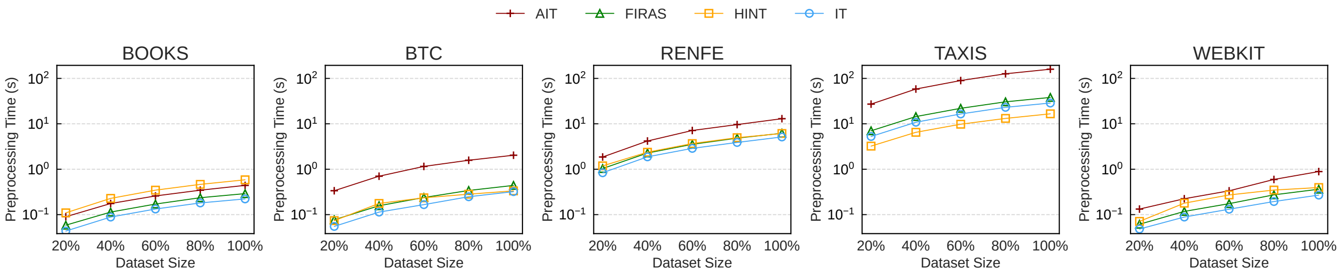

Figure 5: Indexing time varying dataset size

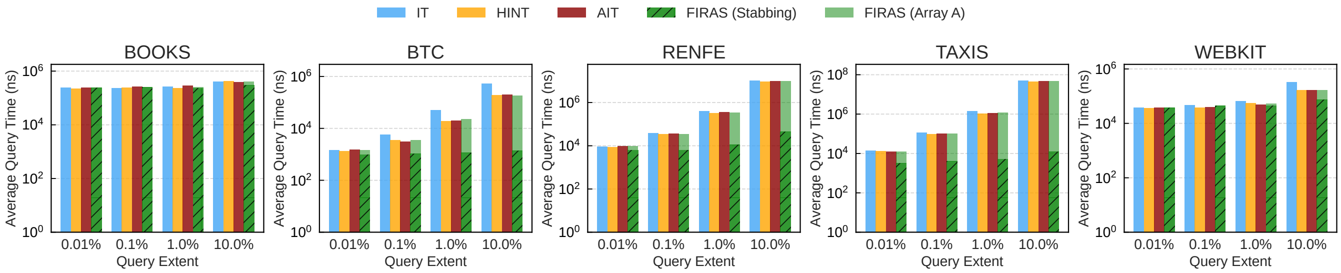

Figure 6: Static data case, range query: FIRAS (Stabbing) and FIRAS (Array A) represent the times for stabbing query processing on IT and 1D range query processing on the sorted array $\mathcal{A}$, respectively.

Table 3: Static data case, pre-processing time (s)

Method	BOOKS	BTC	TAXIS	WEBKIT	RENFE
IT	0.23	0.31	28.98	0.26	5.18
HINT	0.58	0.33	16.40	0.40	6.14
AIT	0.45	2.06	158.06	0.87	13.04
FIRAS	0.29	0.44	37.75	0.35	6.26

Table 4: Static data case, memory usage (MB)

Method	BOOKS	BTC	TAXIS	WEBKIT	RENFE
IT	31	52	2616	44	592
HINT	109	112	3209	106	1517
AIT	70	255	20168	137	3336
FIRAS	47	71	3907	61	888

space requirements. HINT consumes moderately more memory due to interval replication across partitions — an effect particularly evident in BOOKS and WEBKIT datasets, where their long intervals are replicated across multiple partitions. AIT has the highest memory requirements, due to its augmented lists.

5.2.2 Range Queries. Next, we study the range query performance of each method. Figure 6 reports their average query time, varying the query extent. Recall that FIRAS uses the sorted array $\mathcal{A}$ and IT to process range queries, so we break down its time into the time on the sorted array $\mathcal{A}$ and the stabbing time on IT. They are shown as "FIRAS (Array A)" and "FIRAS (Stabbing)" in Figure 6, respectively. Overall, we observe that FIRAS has superior performance to IT and comparable performance to HINT, the state-of-the-art method for range queries, and AIT (without the need for defining its augmented lists). As expected, the average time rises for all methods as the query extent increases, because they need to process more tree nodes (or partitions) and intervals. Nevertheless, this increase is reflected only to the cost of using the array $\mathcal{A}$ on FIRAS; the cost of evaluating stabbing queries on IT is guaranteed to be unaffected when increasing the query extent.

5.2.3 Independent Range Sampling. Figure 7 reports the IRS performance of each method while varying both the query extent and the requested sample size. The numbers over the bars are equal to the average size of a query result. To unveil the details of the observed times, we also show their breakdown into *logging* and *sampling* times. For HINT, this breakdown is omitted, as its total time is dominated by materializing the query results, with the cost of subsequent sampling being minimal. For the other methods, logging time is the cost of determining the range inside every node where results for the range query are found, without, however, materializing these results. For IT, AIT, and FIRAS, sampling time includes the costs of constructing Walker's alias structure, as discussed in Section 3.2, and random sampling.

We observe that FIRAS and AIT consistently deliver the best performance, while AIT is only slightly faster. Compared to FIRAS, which employs the interval tree only to answer a stabbing query and therefore visits $O(\log n)$ nodes, IT follows multiple tree paths and visits a larger number of nodes, which increases the logging cost. As expected, HINT is outperformed by IT in almost all cases (with the exception of BTC); its time is dominated by the high cost of computing the entire range query result R before sampling.

On another point, AIT and FIRAS scale gracefully to the query extent, maintaining low and stable latency even for large query extents. For FIRAS, this is because logging includes two binary searches on the sorted array $\mathcal{A}$ and a stabbing query on IT, none

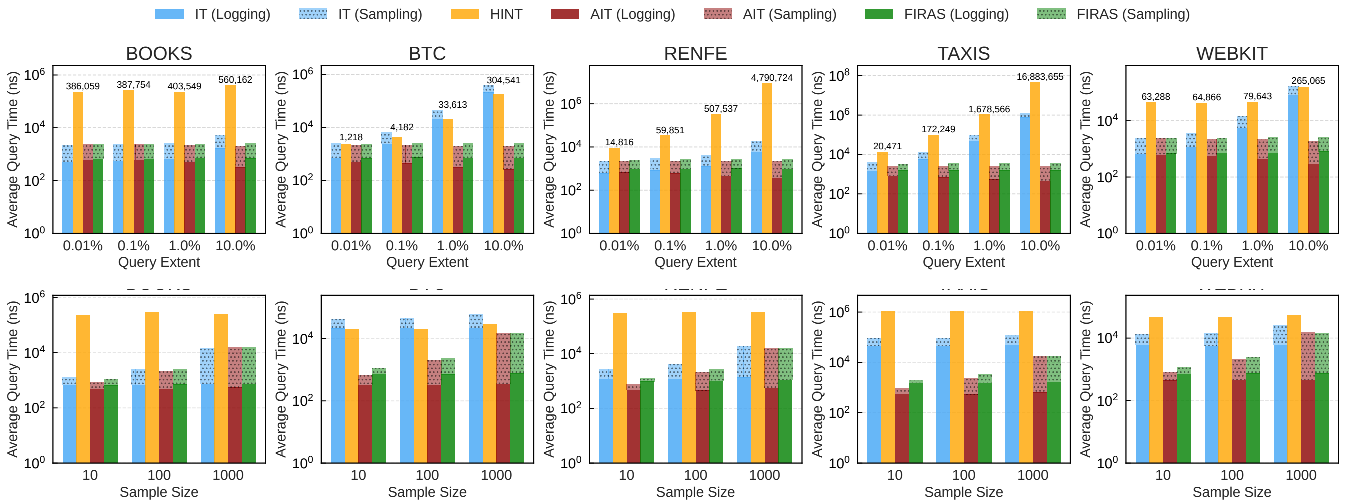

Figure 7: Static data case, independent range sampling (the default query extent and sample size are respectively 1% and 100). Numbers over the bars are equal to the average size of a query result.

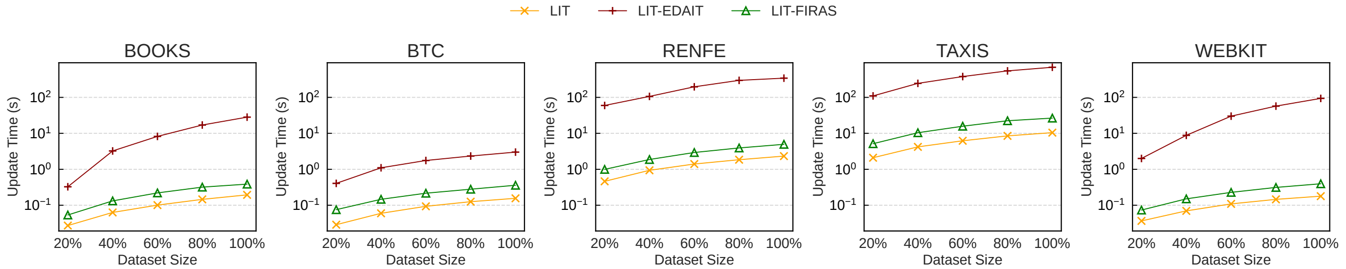

Figure 8: Cumulative update time for evolving X

of which are affected by the increase in the query extent. When increasing the sample size, the average query times of IT, AIT, and FIRAS rise since the sampling cost also grows. In contrast, HINT's time is unaffected since HINT computes R (which is independent of the sample size) before sampling. FIRAS still exhibits comparable query times to the state-of-the-art AIT.

5.2.4 Summary. The experimental results demonstrate that FIRAS practically yields comparable range search (IRS) performance with HINT (AIT), the state-of-the-art method optimized for range search (IRS). This observation further emphasizes the advantage of FIRAS against HINT and AIT: range search and IRS are seamlessly processed and much less space is required, respectively.

5.3 Evolving Interval Data Case

We now proceed to the case of evolving interval data. We evaluated the following methods, measuring their performance in ingesting a stream of intervals and queries. To manage infinite intervals, all methods employ the same data structure, i.e., the sequence of buffers (see Section 4.1), for fair comparison. (Due to this, we use "LIT-" as a prefix of the following methods.)

- LIT [33]: The state-of-the-art method for range queries in evolving interval data. LIT manages closed intervals with a domain-evolving HINT.
- LIT-EDAIT: A modified AIT [4], which we adapted for the dynamic setting for fair comparison. The structure evolves with the domain using the logic of EDIT in Section 4.2.
- LIT-FIRAS: We used EDIT for its interval data structure for closed intervals (see Section 4.2).

5.3.1 Update and Memory Usage Costs. Table 5 reports the total cost of updating the data structures inside each method. Since all methods employ the same structure for infinite intervals, the difference in the observed times is due to the cost of updating the data structures for closed intervals. We observe that LIT exhibits the lowest (best) update time. The update time for LIT-FIRAS is higher because the domain-evolving HINT used by LIT for closed intervals is optimized for updates and has a small number of levels. LIT-EDAIT has the highest update cost, which is dominated by the cost of maintaining its augmented lists.

Figure 8 shows the cumulative update times when X receives 20%, 40%, 60%, 80% and 100% of the intervals. Recall from Lemma 4.1 that the amortized update time of FIRAS is $O(\sqrt{n})$ when D (the domain size) is $O(n)$. In practice, merging buffers, which needs $O(\sqrt{n})$ time, rarely occurs, and the no-merging case requires $O(\log n)$ time (see the proof of Lemma 4.1). Thus, the time for each update is $O(\log n)$ by expectation and, for n intervals, the total update time is $O(n \log n)$, as confirmed experimentally in Figure 8.

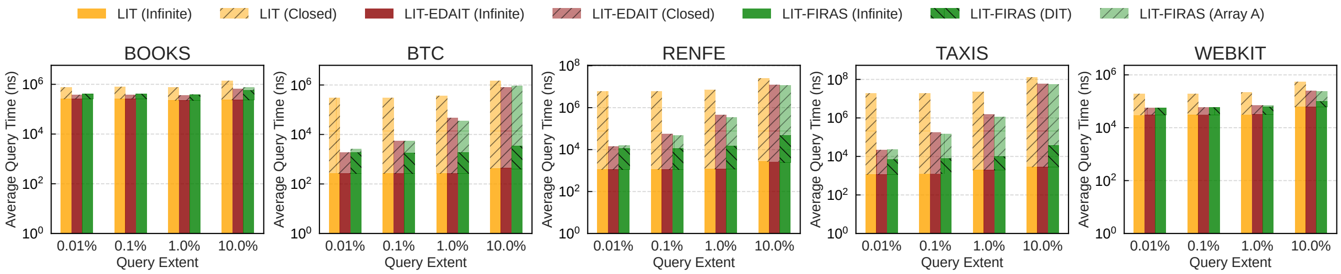

Figure 9: Evolving data case, range query

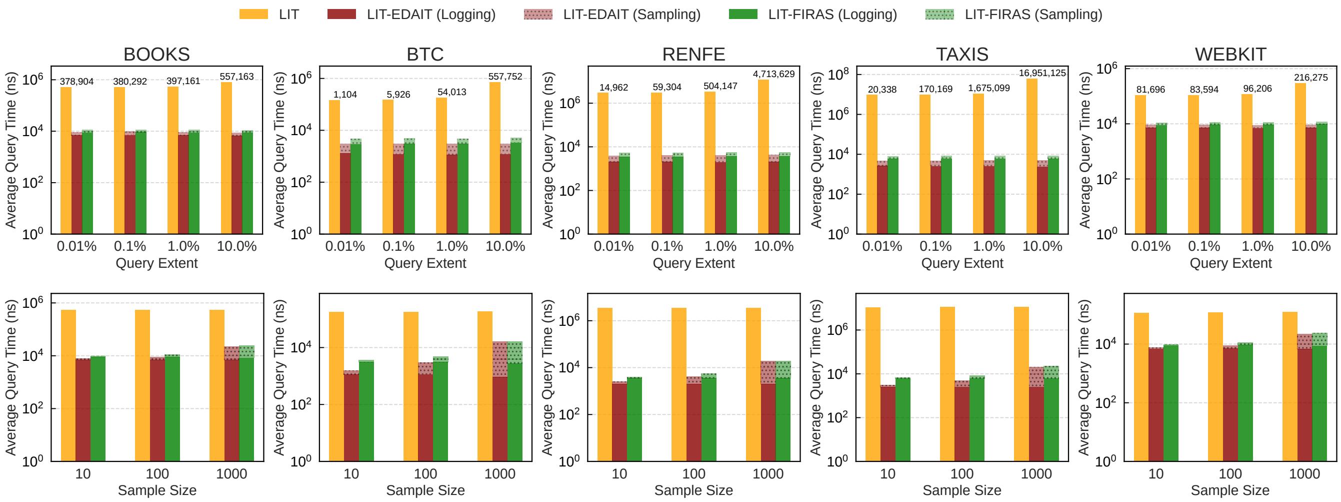

Figure 10: Evolving data case, independent range sampling (the default query extent and sample size are respectively 1% and 100). Numbers over the bars equal to the average size of a query result.

Table 5: Evolving data case, total update time (s)

Method	BOOKS	BTC	TAXIS	WEBKIT	RENFE
LIT	0.18	0.17	11.41	0.18	2.54
LIT-EDAIT	26.15	2.94	589.50	84.21	289.72
LIT-FIRAS	0.37	0.32	26.99	0.33	4.54

Table 6: Evolving data case, memory usage (MB)

Method	BOOKS	BTC	TAXIS	WEBKIT	RENFE
LIT	32	29	1937	33	444
LIT-EDAIT	55	251	19329	161	3381
LIT-FIRAS	47	73	3923	65	888

We also report in Table 6 the memory footprint of each method after handling all updates from the input stream. Similar to the static case, LIT-EDAIT has the highest memory requirements. Contrary to the static case, LIT-FIRAS consumes more space than LIT, because LIT's domain-evolving HINT is optimized for frequent updates and contains fewer levels than HINT in the static case.

5.3.2 Range Query. Figure 9 reports the average query times for varying query extents. As in the static case, we decomposed the LIT-FIRAS time into the time for scanning $\mathcal{A}$ and the time for the stabbing query on EDIT.

We observe that LIT-FIRAS achieves similar performance to LIT-EDAIT but with a fraction of the memory footprint. Intuitively, the query times tend to increase with the query extent, since more tree nodes are visited and longer scans take place in the augmented lists of LIT-EDAIT. LIT performs worse due to its query processing strategy. The reason is that the domain-evolving HINT inside LIT is optimized for updates and has a relatively small number of levels; this has a toll on query times, as larger partitions at the bottom-most level of HINT are accessed and processed by queries.

5.3.3 Independent Range Sampling. Figure 10 reports the average query times for IRS while varying both query extent and sample size. As in the static analysis, the times are decomposed into logging and sampling. For LIT, the logging and sampling phases follow those of HINT in the static case. The logging and sampling phases of LIT-FIRAS and LIT-EDAIT follow those of their static version.

Our tests show that LIT-FIRAS and LIT-EDAIT outperform LIT, while LIT-EDAIT has a slight advantage over LIT-FIRAS. Their query times are unaffected by the query extent, similar to the static case. As expected, LIT performs poorly across all settings because its query time is dominated by the high cost of logging. When the sample size increases, both LIT-FIRAS and LIT-EDAIT scale well; their logging cost remains static while the sampling time increases linearly to the number of samples. LIT's performance, however, remains constant but slow, due to its expensive logging step.

5.3.4 *Summary.* Our conclusions are similar to the static case. Although the update time of LIT-FIRAS is higher than that of LIT, LIT-FIRAS vastly outperforms LIT w.r.t. range and IRS queries. LIT-FIRAS is marginally slower in query processing compared to LIT-EDAIT in some cases; however, LIT-EDAIT consumes a lot of memory, and its update cost is much worse than that of LIT-FIRAS. Overall, *LIT-FIRAS strikes the best tradeoff between update time, memory usage, and query processing time.*

5.3.5 *Discussion about Possible Speed-Up by Parallelization.* Although we focus on single-threaded processing, multi-threading can improve the efficiency of indexing and query processing of FIRAS. For example, parallel sorting can lower the building cost of the sorted array $\mathcal{A}$. For range and IRS queries, FIRAS needs $O(\log^2 n)$ time to determine the relevant interval tree nodes and the ranges $[a_i, b_i]$. As FIRAS identifies $O(\log n)$ nodes to obtain the space of R, the binary searches at these nodes can be done in parallel. Obtaining the query result R can then be done by vectorized scans at each node. Also, each thread (using its own seed) can pick random samples of R in parallel.

In the streaming setting, since the sequence of buffers, $\mathcal{A}$, and EDIT are independent of each other, they can be updated in parallel without locks. When an update and a query arrive at the same time, they can be processed simultaneously by different threads. Based on the idea of optimistic lock coupling [48], assume that each buffer, $\mathcal{A}$, and each node of the EDIT have a version. For threads dealing with a given update, only when updating a buffer, $\mathcal{A}$, or a node of the EDIT, we need an exclusive lock to update the structures. After updating them, they are unlocked, and their versions are incremented. For threads dealing with a query, when they access buffers, $\mathcal{A}$, or a node of the EDIT contributing to R, they check the versions. After processing the query at a buffer, $\mathcal{A}$, or the node, the corresponding versions are checked again. If the versions differ from those checked before, the corresponding threads need to process the query at the corresponding elements.

6 Related Work

6.1 Interval Search

For stabbing queries, segment-tree [35] is a classic data structure that yields $O(\log n + k)$ time. However, it consumes $O(n \log n)$ space. The stabbing problem was theoretically studied decades ago [16, 50, 58], but these studies did not design output-sensitive algorithms and have no practical implementations.

The timeline index [47], which is implemented in SAP HANA [40], maintains intervals based on endpoints. This data structure is a sorted list, with each element being the left- or right-endpoint of an interval. Given a range query q, this list is scanned until the first element having $> q.r$ is reached. The period index [18] is an interval data structure based on a one-dimensional hierarchical grid. The period index is designed to handle not only range queries but also *range-duration queries*, which consider not only overlap but also interval length (i.e., duration). It has been shown in [31, 32] that they are outperformed by the interval tree and HINT.

HINT is also based on a one-dimensional hierarchical grid. The domain of X is hierarchically divided by $m + 1$ levels; each level is uniformly divided into 2^m equi-width partitions. Each interval $x \in X$ is assigned (replicated) to the smallest set of partitions from all levels that collectively cover the interval. This structure enables reporting intervals overlapping a given query, without comparisons, for many cases. The assigned intervals in each partition are further divided into four subdivisions. They guarantee that the same interval cannot be accessed or searched twice by a given query, which is the main property of the subdivisions.

LIT [33] assumes an evolving X and consists of a LiveIndex and a DeadIndex. LiveIndex is a data structure that supports appends, deletions, and range queries; it manages a set of intervals without right-endpoints, whereas the DeadIndex, an extended version of HINT [31], maintains a set of intervals having both endpoints. The DeadIndex can extend its domain and control its depth by being able to delete its bottom level and move intervals to the appropriate partitions of higher levels.

TIDE [60] is a disk-based index for range queries on intervals, which aims at minimizing I/O accesses during range queries and does not support IRS. The RD-index [28, 29] is a 2-dimensional grid structure for range-duration queries. The RD-index is greatly outperformed by HINT for range queries [29], and the RD-index cannot support IRS queries. In addition, the query cost of the RD-index is $O(n \log n)$ (for a fixed block size) [29], which is far worse than the bounds of IT and EDIT. We hence do not consider the RD-index as a component of FIRAS. In [21, 27], some algorithms for batch processing of range queries on HINT are proposed. Recently, Ref. [25] defined relevance queries: their results are ranked based on the relative overlap with the query interval, and those with the largest overlap are returned. Ref. [53] proposed keyword-constrained interval range queries.

Interval joins have also been studied extensively in recent years [22–24, 26, 36, 37, 51, 52]; most of these algorithms are extensions of merge join and plane sweep. Also, some works considered random sampling over joins [6, 34]. We do not present these topics in detail as they address different problems from ours. As discussed in Section 3, Dignös et al. [37] decomposed the interval overlap predicate to reduce an overlap interval join to two *range join queries*. Proposition 3.1 also suggests a query result decomposition, *yet proves a different result.* In particular, Dignös et al. [37] sort (or create a B$^+$-tree with) the left endpoints of each of the two joined relations. The intervals of the other relation are then considered as range queries on the sorted or indexed endpoints. FIRAS employs a totally different strategy: a range query is converted to a stabbing query on an interval data structure and to a 1D range query on the right endpoints of the interval data. Our decomposition improves the complexity of IRS on intervals compared to the previous work [9], and this is a different contribution to [37].

6.2 Independent Range Sampling

IRS in databases was first studied in [49]. Hu et al. [44] proposed an $O(\log n + s)$ "expected" time algorithm on one-dimensional points. Next, in [2], the authors studied the following cases: (i) IRS on weighted one-dimensional points, where each item has a probability (weight) to be sampled, and (ii) IRS with some constraints (e.g., three-dimensional half-space). Afshani and Phillips [1] further improved the latter case by allowing near-linear spaces. The expected and worst times of their algorithm are respectively $O(\log^2 n + s)$ and $o(n)$. Recently, IRS on one-dimensional dynamic points has been studied in [54, 55, 62]. These results are not applicable to our problem, as we deal with intervals, not 1D points.

The problem of IRS on high-dimensional points was studied in [12–15, 43], where random points are sampled from a ball centered at a given query point with a specified radius. KDS [61] is an IRS algorithm for static d-dimensional points. It consumes $O(n)$ space and requires $O(n^{1-1/d} + s)$ time to sample s random points in a given d-dimensional orthogonal range [57]. As seen in Figure 2, intervals can be mapped to two-dimensional points. KDS, hence, can process an IRS query on interval data in $O(\sqrt{n} + s)$ time. This is slower than our result.

7 Conclusion

This paper addresses the problems of range search and independent range sampling on interval data. FIRAS can use any interval data structure; we use the interval tree (IT), which is space-optimal and enables an output-sensitive algorithm for stabbing queries. Thanks to this, we prove that, with $O(n)$ space, FIRAS evaluates range and IRS queries in $O(\log n + k)$ and $O(\log^2 n + s)$ times, respectively. Furthermore, we extended FIRAS to support dynamic (streaming) interval data, and showed that FIRAS has excellent range query and IRS performances with a low update cost. Our extensive experimental results on real-world datasets demonstrate the efficiency of FIRAS on static and streaming interval data.

Acknowledgments

This work is partially supported by AIP Acceleration Research JST (JPMJCR23U2), Adopting Sustainable Partnerships for Innovative Research Ecosystem JST (JPMJAP2328), and Broadening Opportunities for Outstanding young researchers and doctoral students in STrategic areas JST (JPMJBY24A3). It was also supported by project MIS 5154714 of the National Recovery and Resilience Plan Greece 2.0 funded by the European Union under the NextGenerationEU Program.

References

[1] Peyman Afshani and Jeff M. Phillips. 2019. Independent Range Sampling, Revisited Again. In *SoCG (LIPIcs, Vol. 129)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:13. doi:10.4230/LIPICS.SOCG.2019.4

[2] Peyman Afshani and Zhewei Wei. 2017. Independent Range Sampling, Revisited. In *ESA (LIPIcs, Vol. 87)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:14. doi:10.4230/LIPICS.ESA.2017.3

[3] Daichi Amagata. 2023. Diversity Maximization in the Presence of Outliers. In *AAAI*. AAAI Press, 12338–12345. doi:10.1609/AAAI.V37I10.26454

[4] Daichi Amagata. 2024. Independent Range Sampling on Interval Data. In *ICDE*. IEEE, 449–461. doi:10.1109/ICDE60146.2024.00041

[5] Daichi Amagata. 2024. Independent Range Sampling on Interval Data (Longer Version). *CoRR* abs/2405.08315 (2024). arXiv:2405.08315 doi:10.48550/ARXIV.2405.08315

[6] Daichi Amagata. 2025. Random Sampling Over Spatial Range Joins. In *ICDE*. IEEE, 2080–2093. doi:10.1109/ICDE65448.2025.00158

[7] Daichi Amagata and Takahiro Hara. 2016. Diversified Set Monitoring over Distributed Data Streams. In *DEBS*. ACM, 1–12. doi:10.1145/2933267.2933298

[8] Daichi Amagata and Jimin Lee. 2025. Top-k Range Search on Weighted Interval Data. In *SSTD*. ACM, 218–228. doi:10.1145/3748777.3748778

[9] Daichi Amagata, Junya Yamada, Yuchen Ji, and Takahiro Hara. 2024. Efficient Algorithms for Top-k Stabbing Queries on Weighted Interval Data. In *DEXA (Lecture Notes in Computer Science, Vol. 14910)*. Springer, 146–152. doi:10.1007/978-3-031-68309-1_12

[10] Daichi Amagata, Junya Yamada, Yuchen Ji, and Takahiro Hara. 2024. Efficient Algorithms for Top-k Stabbing Queries on Weighted Interval Data (Full Version). *CoRR* abs/2405.05601 (2024). arXiv:2405.05601 doi:10.48550/ARXIV.2405.05601

[11] Kazuyoshi Aoyama, Daichi Amagata, Sumio Fujita, and Takahiro Hara. 2023. Simpler is Much Faster: Fair and Independent Inner Product Search. In *SIGIR*. ACM, 2379–2383. doi:10.1145/3539618.3592061

[12] Martin Aumüller, Sariel Har-Peled, Sepideh Mahabadi, Rasmus Pagh, and Francesco Silvestri. 2021. Fair near neighbor search via sampling. *SIGMOD Rec.* 50, 1 (2021), 42–49. doi:10.1145/3471485.3471496

[13] Martin Aumüller, Sariel Har-Peled, Sepideh Mahabadi, Rasmus Pagh, and Francesco Silvestri. 2022. Sampling a Near Neighbor in High Dimensions - Who is the Fairest of Them All? *ACM Trans. Database Syst.* 47, 1 (2022), 4:1–4:40. doi:10.1145/3502867

[14] Martin Aumüller, Sariel Har-Peled, Sepideh Mahabadi, Rasmus Pagh, and Francesco Silvestri. 2022. Sampling near neighbors in search for fairness. *Commun. ACM* 65, 8 (2022), 83–90. doi:10.1145/3543667

[15] Martin Aumüller, Rasmus Pagh, and Francesco Silvestri. 2020. Fair Near Neighbor Search: Independent Range Sampling in High Dimensions. In *PODS*. ACM, 191–204. doi:10.1145/3375395.3387648

[16] Paul Beame and Faith E. Fich. 2002. Optimal Bounds for the Predecessor Problem and Related Problems. *J. Comput. Syst. Sci.* 65, 1 (2002), 38–72. doi:10.1006/JCSS.2002.1822

[17] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. 1996. An Asymptotically Optimal Multiversion B-Tree. *VLDB J.* 5, 4 (1996), 264–275. doi:10.1007/S007780050028

[18] Andreas Behrend, Anton Dignös, Johann Gamper, Philip Schmiegelt, Hannes Voigt, Matthias Rottmann, and Karsten Kahl. 2019. Period Index: A Learned 2D Hash Index for Range and Duration Queries. In *SSTD*. ACM, 100–109. doi:10.1145/3340964.3340965

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

[20] Michael H. Böhlen, Anton Dignös, Johann Gamper, and Christian S. Jensen. 2017. Temporal Data Management - An Overview. In *Business Intelligence and Big Data*, Esteban Zimányi (Ed.), Vol. 324. Springer, 51–83. doi:10.1007/978-3-319-96655-7_3

[21] Panagiotis Bouros, George Christodoulou, Christian Rauch, Artur Titkov, and Nikos Mamoulis. 2025. Querying Interval Data on Steroids. *IEEE Trans. Knowl. Data Eng.* 37, 10 (2025), 6120–6134. doi:10.1109/TKDE.2025.3597399

[22] Panagiotis Bouros, Konstantinos Lampropoulos, Dimitrios Tsitsigkos, Nikos Mamoulis, and Manolis Terrovitis. 2020. Band Joins for Interval Data. In *EDBT*. OpenProceedings.org, 443–446. doi:10.5441/002/EDBT.2020.53

[23] Panagiotis Bouros and Nikos Mamoulis. 2017. A Forward Scan based Plane Sweep Algorithm for Parallel Interval Joins. *Proc. VLDB Endow.* 10, 11 (2017), 1346–1357. doi:10.14778/3137628.3137644

[24] Panagiotis Bouros and Nikos Mamoulis. 2018. Interval Count Semi-Joins. In *EDBT*. OpenProceedings.org, 425–428. doi:10.5441/002/EDBT.2018.38

[25] Panagiotis Bouros and Nikos Mamoulis. 2025. Relevance Queries for Interval Data. *Proc. ACM Manag. Data* 3, 3 (2025), 206:1–206:26. doi:10.1145/3725343

[26] Panagiotis Bouros, Nikos Mamoulis, Dimitrios Tsitsigkos, and Manolis Terrovitis. 2021. In-Memory Interval Joins. *VLDB J.* 30, 4 (2021), 667–691. doi:10.1007/S00778-020-00639-0

[27] Panagiotis Bouros, Artur Titkov, George Christodoulou, Christian Rauch, and Nikos Mamoulis. 2024. HINT on Steroids: Batch Query Processing for Interval Data. In *EDBT*. OpenProceedings.org, 440–446. doi:10.48786/EDBT.2024.38

[28] Matteo Ceccarello, Anton Dignös, Johann Gamper, and Christina Khnaisser. 2023. Indexing Temporal Relations for Range-Duration Queries. In *SSDBM*. ACM, 3:1–3:12. doi:10.1145/3603719.3603732

[29] Matteo Ceccarello, Anton Dignös, Johann Gamper, and Christina Khnaisser. 2025. Indexing temporal relations for range-duration queries. *Distributed Parallel Databases* 43, 1 (2025), 7. doi:10.1007/S10619-024-07452-6

[30] Reynold Cheng, Dmitri V. Kalashnikov, and Sunil Prabhakar. 2003. Evaluating Probabilistic Queries over Imprecise Data. In *SIGMOD*. ACM, 551–562. doi:10.1145/872757.872823

[31] George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2022. HINT: A Hierarchical Index for Intervals in Main Memory. In *SIGMOD*. ACM, 1257–1270. doi:10.1145/3514221.3517873

[32] George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2024. HINT: a hierarchical interval index for Allen relationships. *VLDB J.* 33, 1 (2024), 73–100. doi:10.1007/S00778-023-00798-W

[33] George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2024. LIT: Lightning-fast In-memory Temporal Indexing. *Proc. ACM Manag. Data* 2, 1 (2024), 20:1–20:27. doi:10.1145/3639275

[34] Binyang Dai, Xiao Hu, and Ke Yi. 2024. Reservoir Sampling over Joins. *Proc. ACM Manag. Data* 2, 3 (2024), 118. doi:10.1145/3654921

[35] Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. 2008. *Computational Geometry: Algorithms and Applications, 3rd Edition*. Springer. doi:10.1007/978-3-540-77974-2

[36] Anton Dignös, Michael H. Böhlen, and Johann Gamper. 2014. Overlap Interval Partition Join. In *SIGMOD*. ACM, 1459–1470. doi:10.1145/2588555.2612175

[37] Anton Dignös, Michael H. Böhlen, Johann Gamper, Christian S. Jensen, and Peter Moser. 2022. Leveraging range joins for the computation of overlap joins. *VLDB J.* 31, 1 (2022), 75–99. doi:10.1007/S00778-021-00692-3

[38] Herbert Edelsbrunner. 1983. A new approach to rectangle intersections part I. *International Journal of Computer Mathematics* 13, 3-4 (1983), 209–219. doi:10.1080/00207168308803364

[39] Erhan Erkut, Yilmaz Ülküsal, and Oktay Yeniçerioglu. 1994. A comparison of *p*-dispersion heuristics. *Comput. Oper. Res.* 21, 10 (1994), 1103–1113. doi:10.1016/0305-0548(94)90041-8

[40] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2011. SAP HANA Database: Data Management for Modern Business Applications. *SIGMOD Rec.* 40, 4 (2011), 45–51. doi:10.1145/2094114.2094126

[41] Thanasis Georgiadis and Nikos Mamoulis. 2023. Raster Intervals: An Approximation Technique for Polygon Intersection Joins. *Proc. ACM Manag. Data* 1, 1 (2023), 36:1–36:18. doi:10.1145/3588716

[42] Thanasis Georgiadis, Eleni Tzirita Zacharatou, and Nikos Mamoulis. 2025. Raster interval object approximations for spatial intersection joins. *VLDB J.* 34, 1 (2025), 8. doi:10.1007/S00778-024-00887-4

[43] Sariel Har-Peled and Sepideh Mahabadi. 2019. Near Neighbor: Who is the Fairest of Them All?. In *NeurIPS*. 13176–13187. https://proceedings.neurips.cc/paper/2019/hash/742141ceda6b8f6786609d31c8ef129f-Abstract.html

[44] Xiaocheng Hu, Miao Qiao, and Yufei Tao. 2014. Independent Range Sampling. In *PODS*. ACM, 246–255. doi:10.1145/2594538.2594545

[45] Christian S. Jensen and Richard T. Snodgrass. 1999. Temporal Data Management. *IEEE Trans. Knowl. Data Eng.* 11, 1 (1999), 36–44. doi:10.1109/69.755613

[46] Georgios Kalamatianos, Georgios John Fakas, and Nikos Mamoulis. 2021. Proportionality in Spatial Keyword Search. In *SIGMOD*. ACM, 885–897. doi:10.1145/3448016.3457309

[47] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter M. Fischer, Donald Kossmann, Franz Färber, and Norman May. 2013. Timeline Index: a Unified Data Structure for Processing Queries on Temporal Data in SAP HANA. In *SIGMOD*. ACM, 1173–1184. doi:10.1145/2463676.2465293

[48] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* 42, 1 (2019), 73–84. http://sites.computer.org/debull/A19mar/p73.pdf

[49] Frank Olken and Doron Rotem. 1989. Random Sampling from B+ Trees. In *VLDB*. Morgan Kaufmann, 269–277. http://www.vldb.org/conf/1989/P269.PDF

[50] Mark H Overmars. 1988. Computational Geometry on a Grid: An Overview. *Theoretical Foundations of Computer Graphics and CAD* (1988), 167–184.

[51] Danila Piatov, Sven Helmer, and Anton Dignös. 2016. An interval join optimized for modern hardware. In *ICDE*. IEEE Computer Society, 1098–1109. doi:10.1109/ICDE.2016.7498316

[52] Danila Piatov, Sven Helmer, Anton Dignös, and Fabio Persia. 2021. Cache-efficient sweeping-based interval joins for extended Allen relation predicates. *VLDB J.* 30, 3 (2021), 379–402. doi:10.1007/S00778-020-00650-5

[53] Christian Rauch and Panagiotis Bouros. 2025. Fast Indexing for Temporal Information Retrieval. *Proc. ACM Manag. Data* 3, 4 (2025), 246:1–246:28. doi:10.1145/3749164

[54] Douglas B. Rumbaugh and Dong Xie. 2023. Practical Dynamic Extension for Sampling Indexes. *Proc. ACM Manag. Data* 1, 4 (2023), 254:1–254:26. doi:10.1145/3626744

[55] Douglas B. Rumbaugh, Dong Xie, and Zhuoyue Zhao. 2024. Towards Systematic Index Dynamization. *Proc. VLDB Endow.* 17, 11 (2024), 2867–2879. doi:10.14778/3681954.3681969

[56] Pierangela Samarati and Latanya Sweeney. 1998. Generalizing Data to Provide Anonymity when Disclosing Information (Abstract). In *PODS*. ACM Press, 188. doi:10.1145/275487.275508

[57] Yufei Tao. 2022. Algorithmic Techniques for Independent Query Sampling. In *PODS*, Leonid Libkin and Pablo Barceló (Eds.). ACM, 129–138. doi:10.1145/3517804.3526068

[58] Mikkel Thorup. 2003. Space Efficient Dynamic Stabbing with Fast Queries. In *STOC*, Lawrence L. Larmore and Michel X. Goemans (Eds.). ACM, 649–658. doi:10.1145/780542.780636

[59] A.J. Walker. 1974. New fast method for generating discrete random numbers with arbitrary frequency distributions. *Electronics Letters* 10 (1974), 127–128. Issue 8. doi:10.1049/el:19740097

[60] Kai Wang, Moin Hussain Moti, and Dimitris Papadias. 2025. TIDE: Indexing Time Intervals by Duration and Endpoint. In *SSTD*. ACM, 207–217. doi:10.1145/3748777.3748785

[61] Dong Xie, Jeff M. Phillips, Michael Matheny, and Feifei Li. 2021. Spatial Independent Range Sampling. In *SIGMOD*. ACM, 2023–2035. doi:10.1145/3448016.3452806

[62] Fangyuan Zhang, Mengxu Jiang, and Sibo Wang. 2023. Efficient Dynamic Weighted Set Sampling and Its Extension. *Proc. VLDB Endow.* 17, 1 (2023), 15–27. doi:10.14778/3617838.3617840

[63] Ying Zhang, Xuemin Lin, Gaoping Zhu, Wenjie Zhang, and Qianlu Lin. 2010. Efficient Rank based KNN Query Processing over Uncertain Data. In *ICDE*. IEEE Computer Society, 28–39. doi:10.1109/ICDE.2010.5447874

[64] Kaiping Zheng, Hongzhi Wang, Zhixin Qi, Jianzhong Li, and Hong Gao. 2017. A survey of query result diversification. *Knowl. Inf. Syst.* 51, 1 (2017), 1–36. doi:10.1007/S10115-016-0990-4