OUKA
The University of Osaka
Institutional Knowledge Archive

Title	Independent Range Sampling on Interval Data
Author(s)	Amagata, Daichi
Citation	2024 IEEE 40th International Conference on Data Engineering (ICDE). 2024, p. 449-461
Version Type	AM
URL	https://hdl.handle.net/11094/106836
rights	© 2024 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.
Note	

The University of Osaka Institutional Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

The University of Osaka

Independent Range Sampling on Interval Data

Daichi Amagata
Graduate School of Information Science and Technology
Osaka University
Japan
amagata.daichi@ist.osaka-u.ac.jp

***Abstract*—Many applications require efficient management of large sets of intervals because many objects are associated with intervals (e.g., time and price intervals). In such interval management systems, range search is a primitive operator for retrieving and analysis tasks. As dataset sizes are growing nowadays, range search results are also becoming larger, which may overwhelm users and incur long computation time. Because applications are usually satisfied with a subset of the result set, it is desirable to efficiently obtain only small samples from the result set. We therefore address the problem of independent range sampling on interval data, which outputs s random samples that overlap a given query interval and are independent of the samples of all previous queries. To efficiently solve this problem theoretically and practically, we propose a variant of an interval tree, namely the augmented interval tree (or AIT), and we show that there exists an exact algorithm that needs $O(n \log n)$ space and $O(\log^2 n + s)$ time, where n is the dataset size. The simple structure of an AIT provides flexible extensions: (i) its time and space complexities respectively become $O(\log^2 n + s)$ expected and $O(n)$ by bucketing intervals and (ii) it can deal with weighted intervals and outputs s weighted random samples in $O(\log^2 n + s \log n)$ time. We conduct extensive experiments on real datasets, and the results demonstrate that our algorithms significantly outperform competitors.**

***Index Terms*—interval data, range search, sampling**

I. Introduction

An interval x is defined as a pair of one-dimensional points and $x = [x.l, x.r]$ ($x.l \leq x.r$), where $x.l$ and $x.r$ respectively are the left- and right-endpoints of x. Many applications require efficient interval data management. For example, in temporal databases, each temporal object is a pair of starting and ending times, and the interval represents active time [1]. Uncertain databases also employ similar data management, because some attribute values of uncertain objects are not fixed, and they are represented as bounding intervals [2]. Moreover, stock and cryptocurrency applications record fluctuations of their prices, so the prices of stocks and cryptocurrency are represented by intervals (e.g., $[\min, \max]$) based on a time unit.

One of the most important operators for interval retrieval and analysis is range query [3]–[5]. Consider a set X of n intervals, and given a query interval $q = [q.l, q.r]$, the range search problem finds a set of all intervals in X overlapping q. This set is represented as $q \cap X = \{x \mid x \in X, q.l \leq x.r \wedge x.l \leq q.r\}$. Some examples that use range queries are as follows.

Ex. 1. Vehicle (e.g., taxi) management systems: Show vehicles that were active between 17:00 and 22:00 a week ago.

Ex. 2. Book management systems or libraries: Collect books that were sold or borrowed in the last month.

Ex. 3. Historical cryptocurrency databases: Show when the price of Bitcoin (BTC) falls in [30,000 40,000] dollars.

The search results of the above queries are easily handled if their sizes are small. If the sizes are large, however, users would be overwhelmed, and analysis tasks may become difficult. In addition, the range search time becomes longer, because range search algorithms incur $\Omega(|q \cap X|)$ time. Dataset sizes are growing nowadays, and $|q \cap X| = \Omega(n)$ in practice, so the output sizes of range queries also increase. Then, simply reporting all intervals overlapping q can yield difficulty in subsequent analysis tasks and long running times. Assume, for example, that a taxi management system wants to visualize active taxis to see their location distribution through a range query q as in Ex. 1. If $|q \cap X|$ is huge (e.g., more than hundreds of thousands), such visualization incurs a long delay [6]. In addition, suppose that we are interested in shopping statistics for each month from 2018 to 2023 on an e-commerce site (e.g., an online bookstore as in Ex. 2) to investigate whether we have some purchase pattern changes before, during, and after the COVID-19 pandemic. It is easy to imagine that there are a significant number of transactions[1] in each month, so collecting them incurs a substantial computational cost.

Motivation. Fortunately, many applications do not require complete result sets but are satisfied with their approximate results (e.g., small subsets) to enable interactive analysis [7], [8]. More specifically, random samples from the result sets are sufficient [6], [9]–[18]. In the above taxi visualization and e-commerce applications, random samples are sufficient to see the distribution and estimate statistics. Also, in cryptocurrency databases, the price fluctuation is usually recorded in a fine-grained manner (e.g., every few minutes). As for Ex. 3, using random samples can reduce redundant data while retaining necessary intervals (to know when). It is also important to notice that retrieving only random samples can significantly reduce query processing time. Then, we see that range *sampling* queries solve the drawbacks of range queries.

Motivated by the above observations, this paper addresses the problem of *independent range sampling (IRS) on interval*

[1]Each transaction is considered as an interval, where its left-endpoint is the time when the transaction started (e.g., the item was purchased online) and its right-endpoint is the time when the transaction was completed (e.g., the time when the item was delivered).

data, which, given a query interval q, retrieves s random samples from $q \cap X$, and the s samples must be independent of all previous query results. (Its formal definition appears in Section II-A.) This paper is the first work that addresses this problem. Note that independence is important, particularly when some statistical property is required [14], like the above sampling examples. Without independence, biased results may be observed, leading to wrong conclusions [6].

Challenge. As the above applications require only s random samples, the running times of IRS queries should be sensitive only to s. That is, the time complexity (and the space complexity) of an IRS algorithm should be $\tilde{O}(s)$, where $\tilde{O}(\cdot)$ hides any polylogarithmic factors. This is a natural requirement of applications that specify output sizes for practical and theoretical efficiency [19].

A straightforward approach to solving the IRS problem on interval data is to employ a state-of-the-art range search algorithm (e.g., HINT^m [3]) to obtain $q \cap X$ and then sample s random intervals from $q \cap X$. Although this approach correctly solves the problem, the merit of range sampling is lost, since $\Omega(|q \cap X|) = \Omega(n)$ time is required. One may come up with an idea of using the IRS algorithm for one-dimensional data [16]. This algorithm stores one-dimensional points in a sorted array and uses two binary searches to find the boundary indices that identify the borders covered by a given query range. Then, simple random sampling from the border range achieves an equal and independent sampling probability. One may consider that this algorithm can solve our problem by using the left- and right-endpoints of intervals, but this cannot solve our problem correctly. For each of the intervals fully covered by the query range, both its left- and right-endpoints are contained in the range. On the other hand, for each of the intervals partially covered by the query range, either its left- or right-endpoint is contained in the range. This means that, to enable equal sampling probability for each interval overlapping the query range, we have to identify whether it is fully covered or not. That is, we need to access all intervals in the range in the worst case, which incurs $O(n)$ time, as with search-based algorithms. Another approach is to prepare samples offline [7], but this trivially violates independence.

In [6], several IRS algorithms for spatial points were proposed. They can solve our problem because intervals can be mapped into a 2-dimensional space [20]. However, even the best algorithm among them returns s random samples in $O(\sqrt{n} + s)$ expected time, which is still sensitive to the dataset size. (In addition, the other algorithms incur $O(n)$ time, which loses the efficiency of random sampling.) To summarize, the existing techniques (including IRS algorithms for one-dimensional and spatial data) cannot efficiently and/or correctly solve the IRS problem on interval data, and achieving an $\tilde{O}(s)$ time algorithm is not trivial.

Contribution. We overcome the above challenge and propose several algorithms that run in $\tilde{O}(s)$ time without losing independence and equal sampling probability. We summarize our contributions below.

TABLE I
COMPARISON OF TIME AND SPACE COMPLEXITIES, WHERE n, $q \cap X$, AND s ARE THE DATASET SIZE, RANGE SEARCH RESULT SET, AND SAMPLE SIZE, RESPECTIVELY. NOTE THAT $O(\cdot)$* SUGGESTS EXPECTATION.

	Time	Space	Weighted?
HINT^m [3]	$\Omega(\|q \cap X\|)$	$O(n)$	✓
KDS [6]	$O(\sqrt{n} + s)$*	$O(n)$	
KDS (weighted)	$O(\sqrt{n} + s \log n)$*	$O(n)$	✓
AIT	$O(\log^2 n + s)$	$O(n \log n)$	
AIT-V	$O(\log^2 n + s)$*	$O(n)$	
AWIT	$O(\log^2 n + s \log n)$	$O(n \log n)$	✓

• *AIT: Augmented Interval Tree* (Section III). We propose AIT, which augments the most famous data structure for interval data, namely Edelsbrunner's interval tree [21]. It is important to note that the original interval tree structure supports efficient processing of stabbing queries but *does not support efficient range-related operators*, e.g., range queries, because $O(n)$ time is incurred in the worst case. Our augmentation overcomes this drawback and derives a new technique that can identify the space where *only* $q \cap X$ exists in $\tilde{O}(1)$ time. Thanks to this technique, we can randomly sample s intervals while preserving the independence, resulting in $\tilde{O}(s)$ time. Furthermore, we show how to efficiently support interval insertions and deletions.

• *AIT-V: AIT with virtual intervals* (Section III-C). Because an AIT needs $O(n \log n)$ space, we next consider achieving $O(n)$ space complexity. This can be done by allowing expected time. We bundle intervals to make "virtual" intervals, and these virtual intervals are maintained by an AIT. We show that this black-box bundling approach (with size constraint) enables $\tilde{O}(s)$ expected time and $O(n)$ space.

• *AWIT: Augmented Weighted Interval Tree* (Section IV). As the second extension of the AIT, we propose AWIT to deal with weighted intervals. This weighted case is also an important setting and has been studied in existing works (with different data formats) [6], [15], [18], [22]. If each interval x_i has a weight $w(x_i)$, its sampling probability should be $\frac{w(x_i)}{\sum_{x_j \in q \cap X} w(x_j)}$. Although achieving this for arbitrary queries is not trivial without accessing all intervals in $q \cap X$, we demonstrate that the AWIT structure keeps $\tilde{O}(s)$ time in this weighted case.

• *Empirical evaluation.* (Section V) We conduct extensive experiments on real-world datasets. The results demonstrate that our algorithms are usually one order of magnitude faster than competitors in non-weighted and weighted interval cases.

Table I compares our theoretical contribution with the existing ones. Since $|q \cap X| = \Omega(n)$ in practice, our algorithms are faster than the range search algorithms. In addition, for sufficiently large n, we have $\log^2 n < \sqrt{n}$, so our algorithms are faster than KDS. These results clarify the superiority and novelty of our algorithms.

II. PRELIMINARY

A. Problem Definition

Let X be a set of n intervals. We use x to denote an interval in X, and x is defined as a pair of its left-endpoint $x.l$ and right-endpoint $x.r$, i.e., $x = [x.l, x.r]$ ($x.l \leq x.r$). Let $x \cap x'$ mean that x overlaps x', i.e., $(x.l \leq x'.r) \wedge (x'.l \leq x.r)$. Similarly, we define $x \cap X$ as $\{x' \,|\, x' \in X, x \cap x'\}$. Now we are ready to formally define the problem of independent range sampling (IRS) on interval data.

PROBLEM 1 (IRS ON INTERVAL DATA). *Given an interval set X, a query interval q, and a sample size s, this problem returns a set S of s intervals, each of which is picked from $q \cap X$ uniformly at random.*

The above definition means that the returned s random samples are independent of the outputs of all previous IRS queries.

Next, if each interval $x \in X$ has a weight $w(x)$, its sampling probability should be proportional to $w(x)$. The IRS problem on weighted interval data is formally defined as follows:

PROBLEM 2 (IRS ON WEIGHTED INTERVAL DATA). *Given an interval set X, a query interval q, and a sample size s, this problem returns a set S of s weighted intervals randomly picked from $q \cap X$. The sampling probability of $x \in q \cap X$ is $\frac{w(x)}{\sum_{x_i \in q \cap X} w(x_i)}$.*

We assume that X is memory resident, as with the existing works that deal with interval data [3], [4], [20], [23]–[28], because recent machines equip memory with large capacity. The objective of this paper is to design theoretically and practically efficient algorithms for the IRS problems on non-weighted and weighted interval data without violating the independent sampling probability.

B. Interval tree

We introduce Edelsbrunner's interval tree [21] since we use it as a building block of our data structure. Henceforth, we call it the interval tree simply. Its structure is essentially similar to the binary tree structure. Each of its nodes u_i consists of the following components:

- c_i: the central point.
- L_i^l: a list containing all intervals x such that $x.l \leq c_i \leq x.r$. The intervals in L_i^l are sorted in ascending order of the left-endpoint.
- L_i^r: a list containing the same intervals in L_i^l. The intervals in L_i^r are sorted in ascending order of the right-endpoint.
- u_i^l: the left child node of u_i. The intervals x' maintained by the sub-tree rooted at u_i^l satisfy $x'.r < c_i$.
- u_i^r: the right child node of u_i. The intervals x' maintained by the sub-tree rooted at u_i^r satisfy $x'.l > c_i$.

We use $L_i^l[j]$ ($L_i^r[j]$) to denote the j-th interval in L_i^l (L_i^r). This paper assumes that the index starts from 1.

Construction. Given a set X of intervals, we first create a root node u_{root}. We compute the central (typically median) point among all left- and right-endpoints in X. We then compute all intervals x such that $x.l \leq c_{root} \leq x.r$ to make L_{root}^l and L_{root}^r. At the same time, we obtain sets $X_l = \{x \,|\, x \in X, x.r < c_{root}\}$ and $X_r = \{x \,|\, x \in X, x.l > c_{root}\}$. We create a left (right) child node of u_{root} from X_l (X_r). The interval tree of X is built in this recursive manner until a given node has $X_l = X_r = \varnothing$.

REMARK 1. *The space complexity and the height of an interval tree are $O(n)$ and $O(\log n)$, respectively. The interval tree structure supports efficient stabbing query processing, and a stabbing query is a special case of the range query q such that $q.l = q.r$. It can be processed in $O(\log n + K)$ time, where K is the output size. However, this structure does not support efficient range query processing. This is trivial: if a query interval q covers X, we need to traverse all nodes, deriving $O(n)$ worst time.*

This remark suggests that it is not trivial to identify the space where only $q \cap X$ exists in $\tilde{O}(1)$ time by using the interval tree structure.

C. Weighted Sampling Methods

Last, we introduce existing weighted sampling methods because our algorithms employ them.

Walker's alias method [29]. Given n objects o_1, ..., o_n associated with weights $w_1, ..., w_n$, this method needs $O(n)$ pre-processing time to build a data structure called *alias* with $O(n)$ space. This alias achieves $O(1)$ time weighted sampling, and the sampling probability of w_i is $\frac{w_i}{\sum_{j=1}^n w_j}$.

Alias building. The alias consists of n cells. The constraints of the alias structure are that (i) each cell can have at most two objects and (ii) the n cells have the pre-defined weight capacity τ, which is based on $\sum_{j=1}^n w_j$. The objects with weights not larger than τ are first assigned to distinct cells, and each of these cells maintains $\langle i, w_i \rangle$ ($w_i \leq \tau$). Then, the remaining objects are assigned to empty cells and/or cells having $\langle i, w_i \rangle$ where $w_i < \tau$. Specifically, o_j with $w_j > \tau$ is maintained by multiple cells, and w_j is distributed to these cells according to their remaining capacities[2].

Sampling. When sampling an object, we first pick one cell uniformly at random. If this cell maintains a single object, we return it as a sample. Otherwise, we generate a random weight in $[0, \tau]$ and pick one of the two objects maintained by the cell based on the generated and their weights.

Cumulative sum method. Given n weighted objects, this method needs $O(n)$ pre-processing time to build an array A with n elements and consumes $O(n)$ space. From this array, we can pick o_i in $O(\log n)$ time with probability $\frac{w_i}{\sum_{j=1}^n w_j}$.

Array building. Each element of A maintains the cumulative sum of the weights. Specifically, $A[j]$ maintains $\sum_{i=1}^j w_i$, so building A is straightforward.

Sampling. We generate a random weight w in $[0, \sum_{j=1}^n w_j]$. If $A[k-1] < w \leq A[k]$, we return k (i.e., o_k) as a sample.

[2]If a given cell is empty, it maintains $\langle j, \tau \rangle$. On the other hand, if it already has $\langle i, w_i \rangle$, it additionally maintains $\langle j, \tau - w_i \rangle$. The cells maintaining o_j have one of these cases, and the sum of their corresponding weights is w_j.

Finding k can be done by a binary search, so $O(\log n)$ time is required for one-time sampling. Notice that k can be returned iff we have $\sum_{j=1}^{k-1} w_j < w \leq \sum_{j=1}^{k} w_j$, and its probability is $(\sum_{j=1}^{k} w_j - \sum_{j=1}^{k-1} w_j)/\sum_{j=1}^{n} w_j = w_k/\sum_{j=1}^{n} w_j$.

III. Non-weighted Interval Case

This section solves Problem 1 and proposes $\tilde{O}(s)$ time algorithms.

Main idea. Our first idea for efficiently solving Problem 1 is to exploit the interval tree structure, particularly the sorted lists of each node. Consider node u_i of an interval tree. The intervals overlapping the central point c_i are maintained by L_i^l and L_i^r. Given a query interval q, the intervals in L_i^l (or L_i^r) overlapping q appear in a sequential manner, since they are sorted. To obtain random samples, it is sufficient to know only the pair of the left and right indices of L_i^l (or L_i^r) that represent an "overlapping range" in this node. This can be done in $O(\log n)$ time with a single binary search, because $|L_i^l| = O(n)$.

Example 1. *Fig. 1 depicts node u_i of an interval tree. It maintains $\{x_1, x_2, x_3, x_4, x_5, x_6\}$, and $L_i^l = [x_1, x_2, x_3, x_4, x_5, x_6]$. Due to the interval tree structure, $x_j.l \leq c_i \leq x_j.r$ ($j \in [1, 6]$). Then, a binary search on L_i^l with a query point $q.r$ can find the boundary such that $L_i^l[j].l \leq q.r < L_i^l[j+1].l$ ($j = 4$ in this example). Consequently, we see that $q \cap L_i^l[j]$ ($j \in [1, 4]$) without enumerating the intervals overlapping q.*

Doing the above operation for each node (which is not pruned) obtains a set of pairs of left and right indices, and this set represents the space where only $q \cap X$ exists. However, we need to traverse $O(n)$ nodes in the worst case when we use the interval tree, see Remark 1. Our second idea is, hence, to bound the number of traversed nodes with the height of an interval tree, i.e., $O(\log n)$. Then, only $O(\log n) \times O(\log n) = \tilde{O}(1)$ time is required to run binary searches for identifying the space where only $q \cap X$ exists. To obtain this desirable time without losing correctness, we augment the interval tree structure. *Finding the space where only $q \cap X$ exists in $\tilde{O}(1)$ time with at most a single binary search for each accessed node is a new result and the main difference* to the existing techniques that can be employed for our problem.

Overview. We build our data structure in the pre-processing phase, and our online algorithm consists of two phases. The first phase collects a set of nodes (of our data structure) maintaining the intervals overlapping q in the way described above. The second phase randomly samples s intervals from the data structure obtained in the first phase.

A. AIT: Augmented Interval Tree

Assume that X is indexed by an interval tree. Given a query interval q, not to miss any intervals that overlap q, we need to access all nodes that possibly maintain such intervals. Then, assume that we now access node u_i such that $q.l \leq c_i \leq q.r$. In this case, we have to traverse both its left and right child nodes, because the sub-trees rooted at these nodes may maintain intervals overlapping q. The interval tree structure cannot bound the number of nodes facing this case, resulting in $O(n)$ time to deal with a range query. This issue is common in other state-of-the-art data structures for interval data, e.g., HINT^m [3]. We eliminate this drawback by extending the interval tree structure, and our structure enables us to face the above case *at most once*.

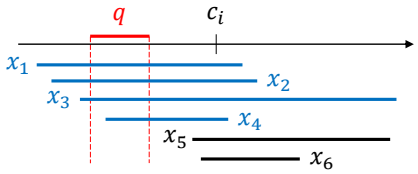


Fig. 1. Illustration of our main idea. Node u_i of an interval tree maintains $\{x_1, x_2, x_3, x_4, x_5, x_6\}$, and $L_i^l = [x_1, x_2, x_3, x_4, x_5, x_6]$. A query q overlaps x_1, x_2, x_3, and x_4.

*1) **Structure**:* We propose AIT (Augmented Interval Tree), a variant of the interval tree, and we augment the node structure of the interval tree. Our augmentation is the key factor for achieving a non-trivial IRS algorithm that has a solid theoretical performance guarantee and avoids the drawback held by the existing structures. Specifically, in addition to the components introduced in Section II-B, each node u_i of an AIT maintains the following components:

- AL_i^l: a list containing all intervals maintained by the sub-tree rooted at u_i. The intervals in AL_i^l are sorted in ascending order of the left-endpoint.
- AL_i^r: a list containing the same intervals in AL_i^l. The intervals in AL_i^r are sorted in ascending order of the right-endpoint.

As with L_i^l and L_i^r, we use $AL_i^l[j]$ ($AL_i^r[j]$) to denote the j-th interval in AL_i^l (AL_i^r). An example of an AIT and its space complexity are as follows:

Example 2. *Fig. 2 illustrates an AIT on a small dataset. Focus on node u_2, and L_2^l, L_2^r, AL_2^l, and AL_2^r are depicted. The sub-tree rooted at u_2 has x_3, x_5, x_7, x_{10}, and x_{11} (their left-endpoints are larger than c_{root}), so they are contained in AL_2^l and AL_2^r.*

Theorem 1 (Space complexity of AIT). *The space complexity of an AIT is $O(n \log n)$.*

Proof. The height of an interval tree is $O(\log n)$, so the height of an AIT is also $O(\log n)$. This fact and the definition of AL^l (AL^r) derive that each interval appears in at most $O(\log n)$ nodes. Then, this theorem is clear. □

*2) **Construction**:* As with the interval tree construction, an AIT is built in a top-down manner and in a pre-processing phase. The procedure is almost the same as that in Section II-B. The only difference is that we create AL^l and AL^r. When we are given $X' \subseteq X$ and create a new node u_i, all intervals in X' are included in AL_i^l and AL_i^r. Note that as long as X is static, the AIT is built only once and accepts arbitrary query intervals and sample sizes.

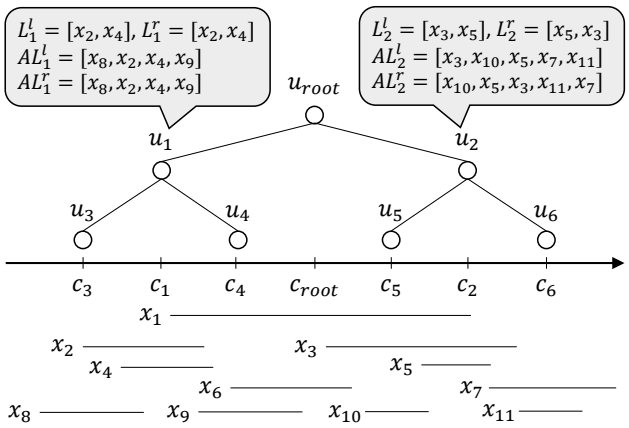


Fig. 2. Illustration of an AIT on a small dataset set $X = \{x_1, x_2, ..., x_{11}\}$. The top part depicts the AIT on X, whereas the bottom part shows x_i ($i \in [1, 11]$).

B. IRS Algorithm for Non-weighted Intervals

1) ***Observation****:* To start with, we elaborate on the observations about, in the AIT, (i) which sorted list should be used and (ii) which nodes should be traversed. Given a query interval q and node u_i of the AIT, this node has one of the following cases:

Case 1: $q.r < c_i$. In this case, the query interval is located to the left of c_i, as shown in Fig. 1. For each $x' \in L_i^l$ such that $x'.l \leq q.r$, we have $x' \cap q$, because $q.r < c_i \leq x'.r$. From the (augmented) interval tree structure, every interval x maintained in the sub-tree rooted at the right child node of u_i has $x.l > c_i$. We can hence prune this sub-tree, and its left child node u_i^l has to be traversed.

Case 2: $c_i < q.l$. The query interval is located to the right of c_i, and for each $x' \in L_i^r$ such that $q.l \leq x'.r$, we have $x' \cap q$ because $x'.l \leq c_i < q.l$. Every interval x maintained in the sub-tree rooted at the left child node of u_i has $x.r < c_i$. This sub-tree can be pruned, and its right child node u_i^r has to be traversed.

Case 3: $q.l \leq c_i \leq q.r$. In this case, q overlaps c_i, so all intervals in L_i^l (and L_i^r) overlap q. As mentioned in Section III-A, both u_i^l and u_i^r have to be traversed in this case. The next (actually last) task we have to address is to find the intervals that overlap $[q.l, c_i)$ and $(c_i, q.r]$ and are maintained at the sub-trees rooted at u_i^l and u_i^r. Thanks to the AIT structure, these intervals are guaranteed to be contained in AL^r (AL^l) of u_i^l and AL^l (AL^r) of u_i^r. That is, different from the existing data structures for intervals, *we need to traverse no more nodes*.

Focus on $u_j = u_i^l$ (the left child node of u_i), and for every $x' \in AL_j^r$, if $q.l \leq x'.r$, then $x' \cap [q.l, c_i)$ because $x'.r < c_i \leq q.r$. Next, focus on $u_k = u_i^r$ (the right child node of u_i), and for every $x' \in AL_k^l$, if $x'.l \leq q.r$, then $x' \cap (c_i, q.r]$ because $q.l \leq c_i < x'.l$.

EXAMPLE 3. *Fig. 3 illustrates an example of case 3. It uses the same intervals in Example 2, so consider the AIT in Fig. 2. The query interval q overlaps c_{root}, so it is guaranteed that*

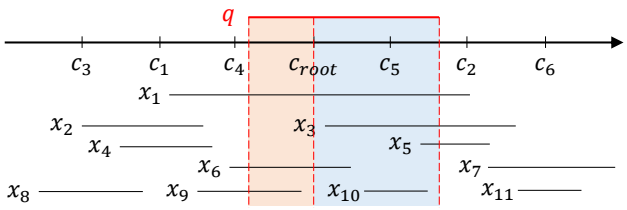


Fig. 3. Illustration of case 3: $q.r \leq c_i \leq q.r$ in Fig. 2 ($c_i = c_{root}$)

q overlaps x_1 and x_6. Then, the remaining task is to find the other intervals existing in the shaded space (or the red and blue spaces). Focus on u_1 (the left child node of u_{root}) The remaining intervals in the red space are certainly contained in AL_1^r. Also, focus on u_2 (the right child node of u_{root}). The remaining intervals in the blue space are also certainly contained in AL_2^l sequentially.

2) ***Algorithm Description****:* Based on the above findings, we design a new algorithm for the IRS problem on interval data, which is summarized in Algorithm 1. Given a query interval q and a sample size s, we access node u_i ($u_i = u_{root}$ at initialization) and do the following operations according to the case where u_i falls in.

Case 1. As in Example 1, we focus on L_i^l and run a binary search with a query point $q.r$ to find the index j such that $L_i^l[j] \leq q.r < L_i^l[j+1]$. From the analysis of case 1 in Section III-B1, each interval $L_i^l[\mathsf{idx}]$ ($1 \leq \mathsf{idx} \leq j$) overlaps q. If $j \geq 1$, we make a node record R_i. This record maintains (a) an integer, which suggests which list is used (0: L^l, 1: L^r, 2: AL^l, and 3: AL^r), and (b) a pair of the left and right indices, which represents a sequence of intervals overlapping q. In this case, $R_i = \langle 0, (1, j) \rangle$. Let $\mathcal{R}$ be a set of node records, and R_i is added to $\mathcal{R}$. After that, we traverse u_i^l (the left child node of u_i) if it exists.

Case 2. We use L_i^r and run a binary search with a query point $q.l$ to find the smallest index j such that $q.l \leq L_i^r[j]$. This is because, from the analysis of case 2 in Section III-B1, we have $L_i^r[\mathsf{idx}] \cap q$ for $j \leq \mathsf{idx} \leq |L_i^r|$. If $j \leq |L_i^r|$, then we make $R_i = \langle 1, (j, |L_i^r|) \rangle$, and this is added to $\mathcal{R}$. If u_i^r (the right child node of u_i) exists, we next traverse it.

Case 3. In this case, all intervals in L_i^l (and L_i^r) overlap q, so we make $R_i = \langle 0, (1, |L_i^l|) \rangle$ and add this to $\mathcal{R}$. Then, we traverse both u_i^l and u_i^r if they exist. As for $u_k = u_i^l$, we use AL_k^r. We run a binary search on AL_k^r with a query point $q.l$ to find the smallest index j such that $q.l \leq AL_k^r[j]$ as with case 2. We make $R_k = \langle 2, (j, |AL_k^r|) \rangle$, and this is added to $\mathcal{R}$. As for $u_{k'} = u_i^r$, we use $AL_{k'}^l$. We run a binary search on $AL_{k'}^l$ with a query point $q.r$ to find the index j such that $AL_{k'}^l[j] \leq q.r < AL_{k'}^l[j+1]$, as in case 1. Also, $R_{k'} = \langle 3, (1, j) \rangle$ is added to $\mathcal{R}$. No more node traversals are required in case 3.

Note that we do not use AL^l and AL^r in cases 1 and 2 to guarantee correctness because, in these cases, even if $x'.l \leq q.r$ for $x' \in AL^l$ (or $q.l \leq x'.r$ for $x' \in AL^r$), it is not

Algorithm 1: IRS on non-weighted interval data

Input: An AIT of X, q, s
Output: S

$S \leftarrow \varnothing$, $\mathcal{R} \leftarrow \varnothing$, $u_i \leftarrow u_{root}$
while 1 **do**
  if $q.r < c_i$ **then**
    $j \leftarrow$ BINARY-SEARCH$(L_i^l, q.r)$ ▷ idx search
    if $j \geq 1$ **then** $\mathcal{R} \leftarrow \mathcal{R} \cup \langle 0, (1, j)\rangle$
    if u_i^l exists **then** $u_i \leftarrow u_i^l$ **else break**
  else if $c_i < q.l$ **then**
    $j \leftarrow$ BINARY-SEARCH$(L_i^r, q.l)$
    if $j \leq |L_i^r|$ **then** $\mathcal{R} \leftarrow \mathcal{R} \cup \langle 1, (j, |L_i^r|)\rangle$
    if u_i^r exists **then** $u_i \leftarrow u_i^r$ **else break**
  else
    $\mathcal{R} \leftarrow \mathcal{R} \cup \langle 0, (1, |L_i^l|)\rangle$
    if u_i^l *exists* **then**
      $u_k = u_i^l$ ▷ go to the left child of u_i
      $j \leftarrow$ BINARY-SEARCH$(AL_k^r, q.l)$
      if $j \leq |AL_k^r|$ **then** $\mathcal{R} \leftarrow \mathcal{R} \cup \langle 2, (j, |AL_k^r|)\rangle$
    if u_i^r *exists* **then**
      $u_k = u_i^r$ ▷ go to the right child of u_i
      $j \leftarrow$ BINARY-SEARCH$(AL_k^l, q.r)$
      if $j \geq 1$ **then** $\mathcal{R} \leftarrow \mathcal{R} \cup \langle 3, (1, j)\rangle$
    break
if $\mathcal{R} = \varnothing$ **then** return $\varnothing$
$\mathcal{A} \leftarrow$ BUILD-ALIAS$(\mathcal{R})$ ▷ Building an alias by Walker's method
while $|S| < s$ **do**
  $R_i = \langle j, (\mathsf{idx}_i^l, \mathsf{idx}_i^r)\rangle \leftarrow$ WEIGHTED-SAMPLING$(\mathcal{A})$
  $\mathsf{idx} \leftarrow$ a random integer in $[\mathsf{idx}_i^l, \mathsf{idx}_i^r]$
  if $j = 0$ **then** $S \leftarrow S \cup \{L_i^r[\mathsf{idx}]\}$
  else if $j = 1$ **then** $S \leftarrow S \cup \{L_i^l[\mathsf{idx}]\}$
  else if $j = 2$ **then** $S \leftarrow S \cup \{AL_i^r[\mathsf{idx}]\}$
  else $S \leftarrow S \cup \{AL_i^l[\mathsf{idx}]\}$
return S

guaranteed that $x' \cap q$.

Now we obtain $\mathcal{R}$, and recall that each node record in $\mathcal{R}$ conceptually represents a sequence of intervals overlapping q. Let idx_i^l and idx_i^r respectively be the left and right indices maintained in R_i. It is important to notice that the sequence size (i.e., $\mathsf{idx}_i^r - \mathsf{idx}_i^l + 1$) is different for each node record. Therefore, to satisfy equal sampling probability for each interval $x \cap q$, simple random sampling is prohibitive and weighted sampling is required. We use Walker's alias method to build an alias of $\mathcal{R}$, where the weight of R_i is $\mathsf{idx}_i^r - \mathsf{idx}_i^l + 1$. Then, we randomly pick a weighted sample (i.e., a node record in $\mathcal{R}$) by using the alias. Assume that this sample is R_i, and we pick an index from $[\mathsf{idx}_i^l, \mathsf{idx}_i^r]$ uniformly at random. From this index and the integer in R_i, we obtain an interval as a random sample. This sampling is repeated s times.

*3) **Analysis**:* We clarify that Algorithm 1 needs $\tilde{O}(s)$ time[3].

THEOREM 2 (TIME COMPLEXITY OF ALGORITHM 1). *The time complexity of Algorithm 1 is $O(\log^2 n + s)$.*

PROOF. Assume that node u_i has case 1 or 2. We need $O(\log n)$ time to obtain the corresponding index in L_i^l or L_i^r, due to the binary search. Recall that the height of the AIT is $O(\log n)$, and we have cases 1 and 2 at most $O(\log n)$ time, so the total time cost of cases 1 and 2 is $O(\log n) \times O(\log n) = O(\log^2 n)$. Next assume that u_i has case 3, and we have this case at most once. The main time cost incurred by case 3 is for the two binary searches on its left and right child nodes, which need $O(\log n)$ time. Hence, computing $\mathcal{R}$ needs $O(\log^2 n)$ time in the worst case.

Now we analyze the time cost for sampling an interval x from $\mathcal{R}$. As seen from the above analysis, we access at most $O(\log n)$ nodes, so $|\mathcal{R}| = O(\log n)$. That is, we need $O(\log n)$ time to build the alias. Section II-C clarifies that sampling a node record from the alias needs $O(1)$ time, and sampling an interval from the node record also needs $O(1)$ time. Given these results, Algorithm 1 needs $O(\log^2 n) + O(\log n) + O(s) = O(\log^2 n + s)$ time. □

We next show that each $x \in q \cap X$ has equal and independent sampling probability.

THEOREM 3 (CORRECTNESS OF ALGORITHM 1). *Algorithm 1 guarantees that, in each sampling iteration, any $x \in q \cap X$ is sampled with probability $\frac{1}{|q \cap X|}$.*

PROOF. We first prove that Algorithm 1 does not miss any intervals $\in q \cap X$ (i.e., no false negatives) and not have any intervals $\notin q \cap X$ (i.e., no false positives). Assume that we have only cases 1 and 2. The pruned sub-trees clearly do not have intervals overlapping q. In addition, for each traversed node u_i, Algorithm 1 evaluates L_i^l or L_i^r, meaning that all of not-pruned intervals are considered. Let idx_i^l and idx_i^r respectively be the obtained left and right indices of L_i^l or L_i^r. The observations in Section III-B1 derive the following three facts. (i) $L_i^l[j]$ or $L_i^r[j]$ overlaps q for $\mathsf{idx}_i^l \leq j \leq \mathsf{idx}_i^r$. (ii) For case 1 and idx such that $\mathsf{idx} > \mathsf{idx}_i^r$, $L_i^l[\mathsf{idx}]$ does not overlap q. (iii) Consider case 2 and idx such that $\mathsf{idx} < \mathsf{idx}_i^l$, $L_i^r[\mathsf{idx}]$ does not overlap q. Therefore, no false positives and negatives are incurred in cases 1 and 2.

We next remove the above assumption: u_i meets case 3. As all intervals in L_i^l (and L_i^r) overlap q, Algorithm 1 does not miss them. Then, Algorithm 1 accesses its left and right child nodes (u_k and $u_{k'}$, respectively). At each of these nodes, we consider all intervals existing in the sub-tree rooted at it, since we use $AL_{k'}^l$ or AL_k^r. Then, Algorithm 1 obtains the left and right indices of $AL_{k'}^l$ and AL_k^r. These indices also satisfy no false positives and negatives, as with cases 1 and 2.

[3]One may have an idea that range trees (by using the 2-dimensional mapping of intervals) can have a similar result. However, range trees generally assume no duplicated coordinates (or the heights become $O(n)$) and cannot early terminate tree traversal even when case 3 is met. The AIT structure does not have these issues.

Consequently, we see that the set $\mathcal{R}$ of node records maintains the space where only $q \cap X$ exists.

Let $X(R_i)$ be a set of intervals that can be sampled by using R_i, and we have $X(R_i) \cap X(R_j) = \varnothing$ for any $R_i, R_j \in \mathcal{R}$. This is because the AIT structure has (a) $L_i \cap L_j = \varnothing$ (L_i can be L_i^l and L_i^r) for any two nodes (u_i and u_j) and (b) $L_i \cap AL_j = \varnothing$ (AL_j can be AL_j^l and AL_j^r) for an arbitrary node u_j and every of its ancestor nodes u_i. Therefore, the sampling probability of a node record R_i is $\frac{\mathsf{idx}_i^r - \mathsf{idx}_i^l + 1}{\sum_{\mathcal{R}} (\mathsf{idx}_j^r - \mathsf{idx}_j^l + 1)} = \frac{\mathsf{idx}_i^r - \mathsf{idx}_i^l + 1}{|q \cap X|}$. The sampling probability of an interval by using R_i is $1/(\mathsf{idx}_i^r - \mathsf{idx}_i^l + 1)$. Consequently, the sampling probability of an interval x from $\mathcal{R}$ (or $q \cap X$) is $1/|q \cap X|$ for any $x \in q \cap X$. □

C. Reducing the Space Complexity

Since the space of an AIT is near linear to n, reducing it to linear to n is desirable. We hence consider how to theoretically guarantee $O(n)$ space. By allowing expected time, we can achieve $O(n)$ space.

Pre-processing. We partition X into disjoint subsets so that each subset has $\Theta(\log n)$ intervals. Therefore, we have $\Theta(\frac{n}{\log n})$ subsets, i.e., $X = \bigcup_{\Theta(\frac{n}{\log n})} X_i$ and, for arbitrary two subsets X_i and X_j ($i \neq j$), $X_i \cap X_j = \varnothing$. (Any partitioning methods can obtain the theoretical result in this section, so we later show our partitioning method.) For each subset, we make a *virtual* interval.

DEFINITION 1 (VIRTUAL INTERVAL). *Given a subset X_i of X, the virtual interval of X_i, v_i, is defined as: $v_i.l = \min_{x \in X_i} x.l$ and $v_i.r = \max_{x \in X_i} x.r$.*

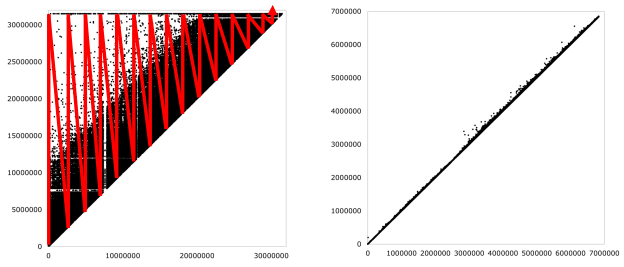
We build an AIT on a set V of virtual intervals, and, since $|V| = \Theta(\frac{n}{\log n})$, we have the following corollary from Theorem 1.

COROLLARY 1. *Given a set V of virtual intervals such that $|V| = \Theta(\frac{n}{\log n})$, the space complexity of the AIT on V is $O(n)$.*

Query processing. Note that the AIT on V is denoted by *AIT-V*. To obtain s random intervals by using the AIT-V, we add the following operations to Algorithm 1. After we sample a virtual interval v_j from R_i, we sample an interval from X_i uniformly at random. (Assume that each X_i has an equal number of intervals, which is easy to hold by adding pseudo-intervals to $X_{\frac{n}{\log n}}$ if necessary.) Let this sample be x, and it is not guaranteed that $x \cap q$. We hence add x into S iff $x \cap q$.

COROLLARY 2 (TIME COMPLEXITY OF OUR IRS ALGORITHM ON AIT-V). *Our algorithm on an AIT-V needs $O(\log^2 + s)$ expected time.*

PROOF. As with Algorithm 1, it needs $O(\log^2 n)$ time to obtain $\mathcal{R}$. Definition 1 guarantees that if $x \in X_i$ has $x \cap q$, $v_i \cap q$. To sample an interval x such that $x \cap q$, we need only a constant number of iterations in expectation [9]. Therefore, we need $O(s)$ expected time to obtain S after obtaining $\mathcal{R}$. □



(a) Book (b) BTC

Fig. 4. Distribution of intervals and rough z-curve (for Book dataset)

It is important to notice that, in AIT-V, each interval x which can be sampled from $\mathcal{R}$ has equal sampling probability. This is trivial from the proof of Theorem 3.

How to partition X. Any disjoint partitioning and Definition 1 provide Corollary 2, but minimizing the number of failure iterations (i.e., sampling an interval $\notin q \cap X$) is crucial for reducing practical time. A reasonable idea is to minimize the difference between v_i and each $x_j \in X_i$. Then, we can formulate an optimization problem which minimizes $\sum_{v_i \in V} \sum_{x_j \in X_i} (|v_i.l - x_j.l| + |v_i.r - x_j.r|)$ such that $|X_k| = \log n$ ($k \in [1, \frac{n}{\log n}]$). This is equivalent to the p-median problem, one of the well-known facility location problems, with capacity constraint, which is NP-hard [30]. (In our case, $v_i \in \mathbb{R}^2$ for each $i \in [1, \frac{n}{\log n}]$.)

We do not solve the above optimization problem but use its concept: the intervals in X_i should be similar to each other. To this end, we employ a pair sort, which sorts X in ascending order of left-endpoint and breaks ties by sorting them in ascending order of right-endpoint. We partition X based on this sort order. Fig. 4 illustrates the distributions of the intervals in the real-world datasets Book and BTC, where each $x = [x.l, x.r]$ is mapped into two-dimensional coordinates $(x.l, x.r)$ (a query interval q is mapped to the rectangle $[-\infty, q.r] \times [q.l, \infty]$). In Fig. 4(a), the red line (roughly) represents the sort order. This figure shows that pair sorting is equivalent to making a z-curve, which maps multi-dimensional points to one-dimensional values while preserving locality.

This approach has two merits: it (i) needs only $O(n \log n)$ additional cost in pre-processing and (ii) functions well in practice for both datasets with large-length intervals (e.g., Book) and with short-length intervals (e.g., BTC). The only possible concern is the partitions with the curves in the former case, because they lose the locality compared with those with no curves. However, each partition contains only $\Theta(\log n)$ intervals, so the impact of the curves is negligible. Our experiments observe that the number of sampling iterations in AIT-V is almost s. For example, when $s = 1000$, it is 1087 (1020) on average in Book (BTC).

D. Updates

We show how to update the AIT structure when an insertion or a deletion is given. (Note that we do not assume frequent

updates, and dynamic intervals with high-frequency updates are not the scope of this work.)

Insertion. Given a new interval, we traverse the AIT nodes in the same way as Algorithm 1, i.e., we traverse the nodes until we have case 3. When a traversed node u_i has case 1 (or 2), we update AL_i^l and AL_i^r and go to its left (or right) child node. If u_i does not have a left (or right) child node, we create it, and the insertion is over. (If the height exceeds $\log n$, we rebuild the AIT to hold the theoretical performance guarantee.) When u_i has case 3, we update L_i^l and L_i^r, and then the insertion is over.

Although the above approach correctly updates the AIT structure, it needs many sorting operations for a single insertion. To alleviate this issue, we use batch insertion, where the batch size is $O(\log^2 n)$. Given an insertion, we add it to an insertion pool and update the AIT only when the pool contains $O(\log^2 n)$ intervals. In this case, we update the AIT similarly to the above one-by-one approach. A difference is that we first insert all new intervals in the pool to the corresponding lists (but sorting is not done at this time) while marking the corresponding nodes, and then we sort the lists in the marked nodes. This approach can reduce the amortized insertion cost while keeping the theoretical performance of Algorithm 1. (When the pool has some new insertions but the AIT is not updated, Algorithm 1 additionally accesses the intervals in the pool, which needs $O(\log^2 n)$ time.)

Deletion. When an interval is removed, we delete it from the corresponding lists of the AIT. The node traversal method is the same as that for the one-by-one insertion. When a node maintains no intervals, we remove it from the AIT. As this deletion does not incur sorting, we do not consider batch updates.

IV. Weighted Interval Case

This section solves Problem 2 and shows that there exists an exact algorithm that needs $\tilde{O}(s)$ time and $\tilde{O}(n)$ space.

Main idea. Our basic idea is to extend the AIT structure and Algorithm 1 for weighted intervals. An intuitive method is to make an alias while considering the weights of $X(R_i)$ for $R_i \in \mathcal{R}$ (recall that $X(R_i)$ is a set of intervals that can be sampled by using R_i). However, to obtain the weights, we need to access all intervals in $X(R_i)$. Since $|X(R_i)|$ can be as large as $O(n)$, this approach incurs $O(n)$ time. Our idea for avoiding this issue is to associate cumulative sums of weights with the sorted lists of each node. This can be prepared in the pre-processing phase. Therefore, for a given query, we need little computational cost to obtain the total weight of the sequence of the intervals defined by idx^l and idx^r. We exploit this idea to achieve an $\tilde{O}(s)$ time algorithm.

A. Augmented Weighted Interval Tree

We extend the AIT structure to efficiently deal with weighted intervals, based on the above idea. This extended AIT is called AWIT (Augmented Weighted Interval Tree). In addition to the components in Section III-A1, each node u_i of an AWIT has the following four arrays:

- W_i^l: an array that maintains the cumulative sums of weights of the intervals in L_i^l. Its j-th element, $W_i^l[j]$, is $\sum_{k=1}^{j} w(L_i^l[k])$.
- W_i^r: an array whose j-th element, $W_i^r[j]$, is $\sum_{k=1}^{j} w(L_i^r[k])$.
- AW_i^l: an array whose j-th element, $AW_i^l[j]$, is $\sum_{k=1}^{j} w(AL_i^l[k])$.
- AW_i^r: an array whose j-th element, $AW_i^r[j]$, is $\sum_{k=1}^{j} w(AL_i^r[k])$.

The above definition suggests that each node of the AWIT needs approximately double-space of that of the AIT, so Theorem 1 derives the following corollary:

Corollary 3 (Space complexity of AWIT). *The space complexity of an AWIT is $O(n \log n)$.*

The main merit of the above arrays is that only $O(1)$ time is needed to obtain the total weight of intervals which can be sampled from R_i. Hence, we can still build an alias of $\mathcal{R}$ in $O(\log n)$ time. Furthermore, when we sample a random weighted interval from R_i, we do not need to build any data structures.

Discussion. Since the above four arrays are based on the sorted intervals in the corresponding lists, even a single insertion/deletion may incur significant updates in the four arrays. We therefore do not consider updates in Section IV, and designing a data structure that supports efficient IRS query processing and updates on weighted interval data is a future work.

In Problem 2, the sampling probability of a weighted interval is defined based on $q \cap X$. This suggests that no false positives are allowed, so the idea of AIT-V (see Section III-C) is not available.

B. IRS Algorithm for Weighted Intervals

Our IRS algorithm for weighted intervals follows Algorithm 1. The differences are summarized below:

- We use an AWIT.
- Consider building an alias (line 23). For each $R_i \in \mathcal{R}$, we use $W_i^l[\mathsf{idx}_i^r]$, $W_i^r[\mathsf{idx}_i^r] - W_i^r[\mathsf{idx}_i^l - 1]$, $AW_i^r[\mathsf{idx}_i^r] - AW_i^r[\mathsf{idx}_i^l - 1]$, or $AW_i^l[\mathsf{idx}_i^r]$ as the weight of R_i when we use L_i^l, L_i^r, AL_i^l, or AL_i^r, respectively.
- When sampling a random idx from $[\mathsf{idx}_i^l, \mathsf{idx}_i^r]$ (line 26), we use the cumulative sum method (see Section II-C) with $(\mathsf{idx}_i^l, \mathsf{idx}_i^r)$ and the corresponding array.

One may consider that the last difference can be replaced with Walker's alias method. If we do this, we need to build an alias on R_i, which again needs to access all intervals in $X(R_i)$. As described before, this is prohibitive. On the other hand, the cumulative sum method does not need to build any data structures online, so we can avoid the $O(n)$ time. From this method and Theorem 2, we have:

Corollary 4 (Time complexity of our algorithm on AWIT). *The time complexity of our IRS algorithm for weighted intervals is $O(\log^2 n + s \log n)$ time.*

From Theorem 3 and the cumulative sum method, moreover, it is straightforward to see that, in the above algorithm, the sampling probability of any $x \in q \cap X$ is $\frac{w(x)}{\sum_{x' \in q \cap X} w(x')}$.

Now we have introduced our theoretical results in this paper and the flexible extensions of the AIT structure. In the next section, we show that the efficiencies of our algorithms are guaranteed not only theoretically but also practically.

V. Experiment

This section reports our experimental results. All experiments were conducted on Ubuntu 22.04 LTS machine with 2.2GHz Intel Core i9-13950HX processor and 128GB RAM.

A. Setting

Datasets. We used four real-world large datasets, Book [31], BTC [32], Renfe [33], and Taxi [34]. Table II shows the statistics of each dataset. Book contains the borrowing periods of books in Aarhus libraries. BTC is a set of historical price intervals of Bitcoin, and we used low and high prices as the left- and right-endpoints, respectively. Renfe and Taxi are respectively Spanish rail and NYC taxi trip data. For Renfe (Taxi), we used departure (pick-up) time and arrival (drop) time as the left- and right-endpoints, respectively. When we considered weighted intervals, we assigned a random weight in $[1, 100]$ to each interval.

Queries. We generated 1,000 queries for each experiment. The left-endpoint of a query was chosen from the corresponding domain uniformly at random, and its interval length was 8% of the domain size by default, which was used to set the right-endpoint. In addition, $s = 1,000$ by default [6].

Evaluated algorithms. We used the following algorithms.

- Interval tree [21]: This algorithm first searches for all intervals overlapping q and randomly samples s intervals from them.
- HINT^m [3], [4], [31]: This took the same approach as Interval tree. (Literatures [3], [4] show that HINT^m significantly outperforms the other state-of-the-art interval search algorithms [5], [35], so we did not consider these algorithms.)
- KDS [6]: This is the state-of-the-art spatial independent range sampling algorithm. (Literature [6] reports that the other algorithms proposed in [6] are outperformed by KDS.)
- AIT: Our algorithm on an AIT (Section III-B).
- AIT-V: Our algorithm on an AIT-V (Section III-C).
- AWIT: Our algorithm on an AWIT (Section IV-B).

The above algorithms were single-threaded, implemented in C++, and compiled by g++ 11.3.0 with -O3 flag[4]. In the case where intervals were non-weighted, we evaluated Interval tree, HINT^m, KDS, AIT, and AIT-V. In the case where intervals were weighted, we evaluated Interval tree, HINT^m, KDS, and AWIT. Actually, KDS cannot solve Problem 2 correctly, because it incurs false positives and does not guarantee $\frac{w(x)}{\sum_{x_i \in q \cap X} w(x_i)}$ sampling probability. We used it to highlight the time difference between AWIT and KDS.

[4]The source codes of our algorithms are available at https://github.com/amgt-d1/IRS-interval

TABLE II
Dataset statistics

Dataset	Book	BTC	Renfe	Taxi
Cardinality	2,295,260	2,538,921	38,753,060	106,685,540
Domain size	31,507,200	6,876,400	52,163,400	79,901,357
Min length	3,600	1	1,320	1
Med. length	1,458,000	937	9,120	663
Max length	31,406,400	547,077	44,700	2,618,881

TABLE III
Pre-processing time [sec] (non-weighted case)

Dataset	Book	BTC	Renfe	Taxi
Interval tree	1.45	2.93	52.62	147.19
HINT^m	0.60	0.20	3.26	4.67
KDS	2.15	3.43	36.16	210.36
AIT	3.02	7.00	103.52	274.02
AIT-V	0.26	0.28	3.91	9.40

TABLE IV
Memory usage [GB] (non-weighted case)

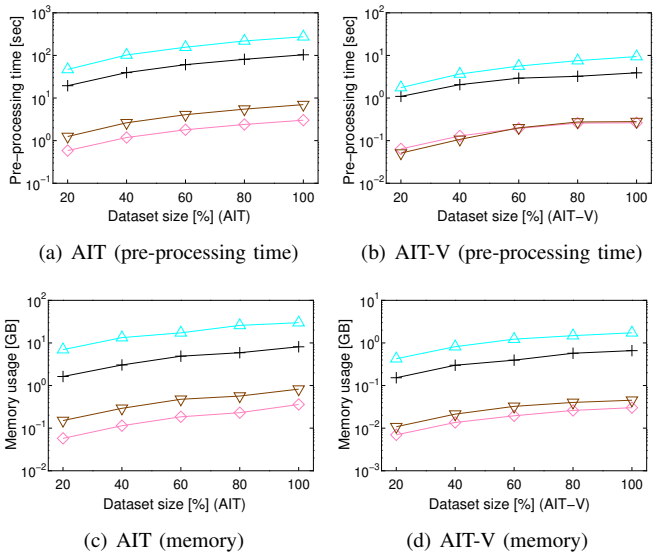
Dataset	Book	BTC	Renfe	Taxi
Interval tree	0.17	0.22	2.26	6.27
HINT^m	0.10	0.06	0.53	1.29
KDS	0.29	0.32	4.84	13.34
AIT	0.30	0.78	8.12	29.88
AIT-V	0.03	0.05	0.66	1.73

B. Result on Non-weighted Intervals

Pre-processing time and memory usage. Table III shows the pre-processing time of each algorithm. AIT needs longer pre-processing times than the others. However, it is acceptable since pre-processing is done only once, and less than 5 minutes is needed even for about 100 million intervals. AIT-V alleviates pre-processing time, since it uses only $\frac{n}{\log n}$ (virtual) intervals. Table IV shows the memory usage of each algorithm, and the result is similar to the pre-processing time. Although AIT needs about 30GB for Taxi (about 100 million intervals), this is easy to fit into modern main memory. If this can be a concern, AIT-V is preferable because it consumes less memory (and yields fast query processing time, as shown later).

Fig. 5 depicts how the pre-processing times and memory usages of our algorithms scale to the dataset size. (We varied the dataset size by random sampling, and plots are log-scale.) We see that the pre-processing time of each of our algorithms scales linearly to n. Moreover, Fig. 5(c) shows that the memory usage of AIT also scales linearly to n, which suggests that its practical space performance is better than its theoretical guarantee. As AIT-V requires $O(n)$ space theoretically, its practical performance follows this result, as shown in Fig. 5(d).

Comparison with existing techniques. We next investigate the query processing performance of each algorithm. Tables V and VI respectively show the average time to compute the candidate (i.e., $|q \cap X|$ for Interval tree and HINT^m, $\mathcal{R}$ for our algorithms, and a superset of $\cup_{\mathcal{R}} X(R_i)$ for KDS)



(a) AIT (pre-processing time) (b) AIT-V (pre-processing time)

(c) AIT (memory) (d) AIT-V (memory)

Fig. 5. Pre-processing time [sec] and memory usage [GB] of AIT and AIT-V. ◇, ▽, +, and △ respectively show the result on Book, BTC, Renfe, and Taxi.

TABLE V
CANDIDATE COMPUTATION TIME [MICROSEC]

Dataset	Book	BTC	Renfe	Taxi
Interval tree	4353.58	3345.17	76304.50	177287.52
HINT^m	4115.27	2183.65	34264.49	131061.57
KDS	105.29	16.37	9.40	44.24
AIT	0.83	0.37	1.20	2.08
AIT-V	0.02	0.01	0.94	1.01

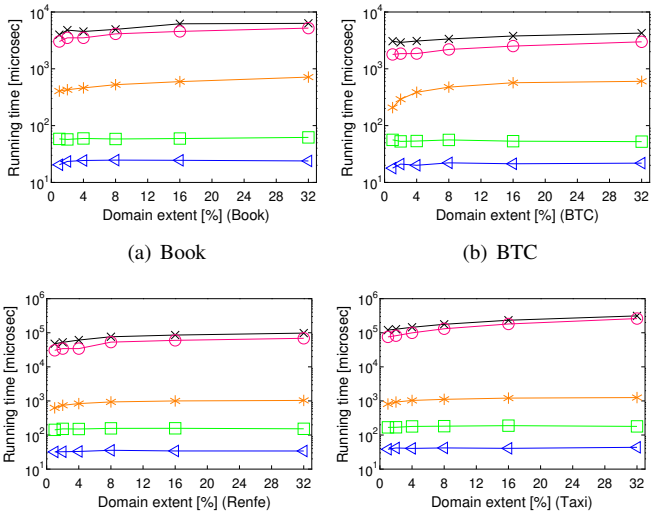
and the average sampling time. Interval tree and HINT^m incur significant time to compute the candidate, since they need to search all intervals in $q \cap X$. After they obtain $q \cap X$, they run simple random sampling, so their sampling times are the shortest. KDS alleviates the candidate computation time but requires the longest sampling time, because it needs to use $O(\sqrt{n})$ nodes.

AIT and AIT-V are much better balanced w.r.t. candidate computation and sampling times than the others. They can obtain $\mathcal{R}$ quite fast, and their sampling times are also short. For example, on Taxi, AIT is about 3131 (51) times faster than HINT^m (KDS) w.r.t. total running time (the sum of candidate computation and sampling times). AIT-V requires a longer sampling time than AIT, because it needs to evaluate whether a sampled interval x has $x \cap q$, whereas AIT does not need this evaluation. This result suggests the trade-off relationship between time and memory usage.

Impact of query interval length. We study the impact of range size, i.e., $q.r - q.l$. Fig. 6 shows the average running times with different query interval lengths (that are equal to the domain extent). Interval tree and HINT^m need longer running time as the domain extent (query interval size) becomes larger. This is trivial, as larger domain extents provide larger $|q \cap X|$, and they incur $\Omega(|q \cap X|)$ time. KDS also needs a bit longer time with the increase of query interval length, which

TABLE VI
SAMPLING TIME [MICROSEC] (NON-WEIGHTED CASE). ALIAS BUILDING TIME IS INCLUDED.

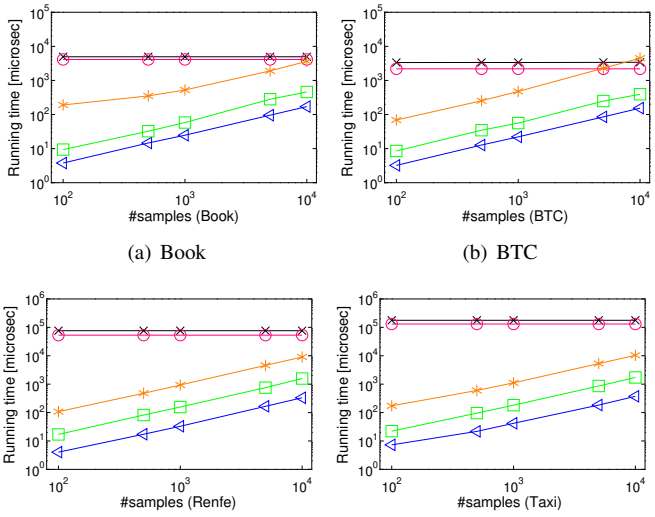
Dataset	Book	BTC	Renfe	Taxi
Interval tree & HINT^m	4.79	7.39	19.81	27.43
KDS	420.13	459.70	925.84	1070.09
AIT	23.88	21.74	35.68	39.77
AIT-V	58.14	56.00	155.93	180.95



(a) Book (b) BTC

(c) Renfe (d) Taxi

Fig. 6. Impact of domain extent (or query interval length) in non-weighted case. × shows Interval tree, ○ shows HINT^m, ∗ shows KDS, ◁ shows AIT, and □ shows AIT-V.



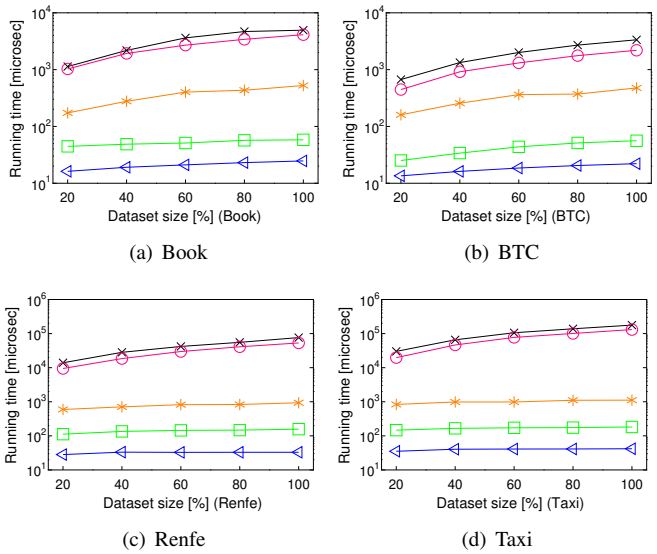
(a) Book (b) BTC

(c) Renfe (d) Taxi

Fig. 7. Impact of sample size in non-weighted case. × shows Interval tree, ○ shows HINT^m, ∗ shows KDS, ◁ shows AIT, and □ shows AIT-V.

is consistent with the result in [6]. On the other hand, our algorithms are not affected by query interval length, and this result confirms the clear advantage of our algorithms against the competitors.

Impact of sample size. Next, we investigate the impact of



(a) Book (b) BTC

(c) Renfe (d) Taxi

Fig. 8. Impact of dataset size in non-weighted case. × shows Interval tree, ○ shows HINT^m, ∗ shows KDS, ◁ shows AIT, and □ shows AIT-V.

TABLE VII
AMORTIZED UPDATE TIME OF AIT [MILLISEC]

Dataset	Book	BTC	Renfe	Taxi
Insertion	448.18	894.44	2283.23	6312.70
Batch insertion	3.01	2.14	5.25	10.43
Deletion	2.23	3.24	31.58	90.38

sample size s, which is shown in Fig. 7. Interval tree and HINT^m are not affected by s, since their dominant costs are derived from computing $q \cap X$, as shown in Tables V and VI. As with our theoretical analysis, the running times of our algorithms (and KDS) are linear to s. Even when many samples (e.g., $s = 10,000$) are required, our algorithms are still faster than the others, while the performance of KDS can be worse than the search-based algorithms.

Impact of dataset size. Fig. 8 reports the scalability to the dataset size. The results of Interval tree and HINT^m suggest that $|q \cap X| = \Omega(n)$. Our algorithms need only tens of microseconds and are clearly not sensitive to the dataset size. The importance of designing an $\tilde{O}(s)$ time algorithm can be seen from this result.

Update time. We introduce the results of AIT update experiments. For the insertion test, we first built an AIT on $n - 5000$ intervals and inserted the remaining 5000 intervals into the AIT. For the deletion test, we built an AIT on X and then removed 5000 intervals from the AIT. Table VII shows the amortized update time per insertion/deletion. One-by-one insertion incurs a long update time even for a single insertion, whereas the batch insertion substantially reduces the update time. Also, a deletion can be handled quickly, suggesting the usefulness of the traversal approach in Algorithm 1.

C. Result on Weighted Intervals

Pre-processing time and memory usage. Tables VIII shows the pre-processing times and memory usages of AWIT. As

TABLE VIII
PRE-PROCESSING TIME [SEC] AND MEMORY USAGE [GB] OF AWIT

Dataset	Book	BTC	Renfe	Taxi
Pre-processing time	3.15	6.07	109.86	282.81
Memory usage	0.44	1.13	12.29	46.15

TABLE IX
SAMPLING TIME [MICROSEC] (WEIGHTED CASE). ALIAS BUILDING TIME IS INCLUDED.

Dataset	Book	BTC	Renfe	Taxi
Interval tree & HINT^m	6594.67	6593.22	122169.91	389509.09
KDS	1307.50	1442.94	1917.36	2101.71
AWIT	136.39	134.06	347.94	446.72

with the results on non-weighted intervals, AWIT does not consume long pre-processing times, and only a little additional cost is incurred. This is also the case for the memory consumption of AWIT.

Comparison with Interval tree, HINT^m, and KDS. We next study the query processing time, and Table IX shows the average time to sample s intervals. Note that the candidate computation times are the same as in Table V. Interval tree and HINT^m incur substantial sampling times, different from the results in Table VI. To correctly enable weighted random sampling, they need to build an alias after $q \cap X$ is obtained. Building it requires $O(|q \cap X|)$ time, so we have this result.

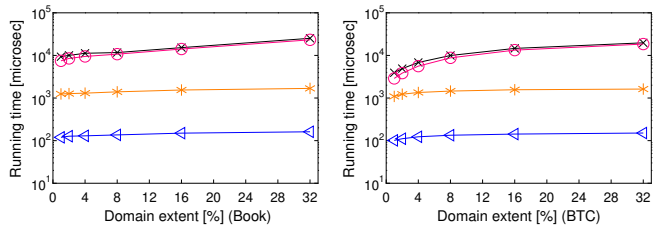
On the other hand, AWIT does not have such a drawback, and they defeat the search-based algorithms and KDS w.r.t. both candidate computation and sampling times. The sampling time of AWIT is longer than those in the non-weighted case. This observation is derived from the $O(\log n)$ factor in sampling a weighted random interval.

Impact of query interval length. Fig. 9 shows the impact of query interval length (domain extent). Although it is slight, our algorithms are affected by query interval length (e.g., see Fig. 9(d)). This is derived from the cumulative sum method. As large query interval lengths yield large $\mathsf{idx}^r - \mathsf{idx}^l$, the binary search cost in the cumulative sum method slightly increases.

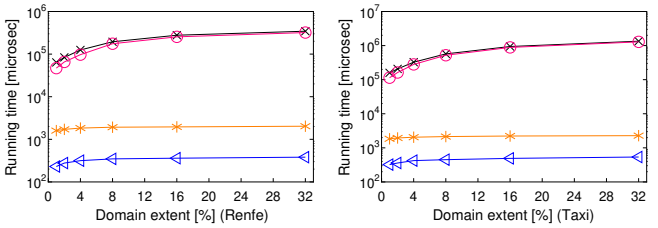
Impact of dataset size. We show the scalability of each algorithm in Fig. 10. Our algorithms need $\tilde{O}(s)$ time, so they again are not sensitive to the dataset size, as with the results in Fig. 8. We omit the impact of sample size, because it essentially yields the same result as that in Fig. 7 and the page space is limited.

VI. RELATED WORK

Range search on interval data. Due to the importance of the interval search problem, data structures for efficient search have been developed. Perhaps, the most famous data structure is interval tree [21], and we used this structure as our building block. In the background of temporal databases, timeline index [35] was proposed, and it is implemented in SAP-HANA [36]. Period index [5] was also devised in the temporal database background. Recently, Christodoulou et al. proposed HINT^m [3], a hierarchical interval index for range queries. (It is a heuristic algorithm and has no theoretical time bound for

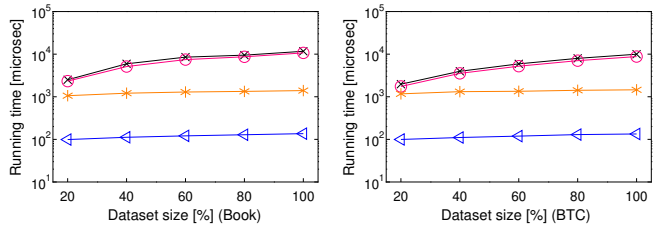


(a) Book (b) BTC

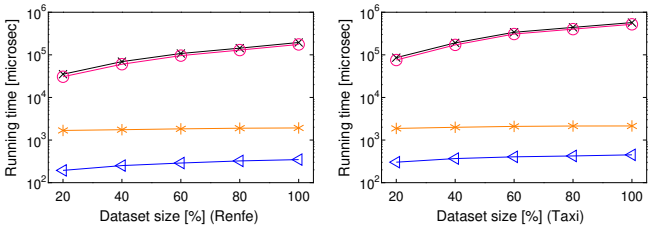


(c) Renfe (d) Taxi

Fig. 9. Impact of domain extent (or query interval length) in weighted case. × shows Interval tree, ○ shows HINT^m , ∗ shows KDS, and ◁ shows AWIT.



(a) Book (b) BTC



(c) Renfe (d) Taxi

Fig. 10. Impact of dataset size in weighted case. × shows Interval tree, ○ shows HINT^m , ∗ shows KDS, and ◁ shows AWIT.

range queries.) This structure is designed so that it can (i) achieve comparison-free, (ii) easily adapt to the distribution of a given dataset (e.g., sparsity), and (iii) exploit hardware-aware optimization. In [3], [4], it is empirically shown that HINT^m outperforms the interval tree, the timeline index, and the period index. We therefore used HINT^m as a competitor, and Section V clarifies that even the state-of-the-art range search algorithm cannot efficiently solve the IRS problem.

Other operations on interval data. Segment-tree [37] is also a famous data structure. It needs $O(n \log n)$ space and supports $O(\log n + K)$ time stabbing query, where K is the output size. (This structure does not support efficient range search.) For weighted intervals, literatures [20], [38], [39] addressed the problem of finding k intervals with the largest weight among the intervals stabbing a given query. The interval join problem, which finds all pairs of intervals overlapping each other, has also been studied [23]–[25], [27], [28], [40], [41]. Given a set X of intervals, literature [26] tackled the problem of disjoint partitioning X for efficient interval join and aggregation. Because the objective of this partitioning is different from ours in Section III-C, the technique proposed in [26] is not available for our problem.

IRS on other data. The IRS problem has been extensively studied, and its usefulness has been recognized since the 1980s [17]. The importance of "independent samples" has spread from [16], which solved the IRS problem on dynamic one-dimensional data and external-memory-resident data. Its weighted version has recently been addressed in [22]. Recall that Section I has clarified the IRS algorithm for one-dimensional data cannot correctly solve our problem.

The IRS problem on multi-dimensional data has been addressed as well. Afshani and Wei [18] considered two- and three-dimensional points, whereas Afshani and Phillips [15] assumed three-dimensional weighted points. These works provided theoretical results for specific queries (e.g., three-dimensional half-space queries [15]), which are different from range queries on interval data. As a practically efficient algorithm, Xie et al. proposed KDS [6], which solves the IRS problem on d-dimensional Euclidean points in $O(n^{1-\frac{1}{d}} + s)$ expected time. Since intervals can be mapped to a two-dimensional Euclidean space and query intervals can be mapped to orthogonal ranges [20], we used KDS as a competitor and demonstrated that our interval-specific data structures provide much faster candidate computation and sampling than KDS theoretically (see Table I) and empirically (cd. Section V).

The above works considered low-dimensional points, and recent works [10]–[12], [42], [43] addressed the IRS problem on high-dimensional spaces. These works consider sampling a single random object with a distance at most a threshold from a given query. They show that such sampling can be done in sub-linear (expected) time to n with high probability [12] or approximate independence probabilities [10]. This setting is totally different from ours, and our theoretical results do not have this probability condition and approximation.

VII. Conclusion

We addressed the problems of independent range sampling on non-weighted and weighted interval data. To theoretically and practically solve these problems efficiently, we proposed some variants of interval tree and $\tilde{O}(s)$ time algorithms, where s is the number of samples. We demonstrated that, for sufficiently large datasets, these algorithms are theoretically faster than existing techniques that can be used for the problems. In addition, from extensive experiments, we confirmed that our algorithms outperform competitors by large margins.

Acknowledgements

This work was partially supported by AIP Acceleration Research JPMJCR23U2 and Adopting Sustainable Partnerships for Innovative Research Ecosystem JPMJAP2328, JST.

REFERENCES

[1] S. Nishio, D. Amagata, and T. Hara, "Lamps: Location-aware moving top-k pub/sub," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 1, pp. 352–364, 2022.

[2] Y. Zhang, X. Lin, G. Zhu, W. Zhang, and Q. Lin, "Efficient rank based knn query processing over uncertain data," in *ICDE*, 2010, pp. 28–39.

[3] G. Christodoulou, P. Bouros, and N. Mamoulis, "Hint: A hierarchical index for intervals in main memory," in *SIGMOD*, 2022, pp. 1257–1270.

[4] ——, "Hint: a hierarchical interval index for allen relationships," *The VLDB Journal*, pp. 1–28, 2023.

[5] A. Behrend, A. Dignös, J. Gamper, P. Schmiegelt, H. Voigt, M. Rottmann, and K. Kahl, "Period index: A learned 2d hash index for range and duration queries," in *SSTD*, 2019, pp. 100–109.

[6] D. Xie, J. M. Phillips, M. Matheny, and F. Li, "Spatial independent range sampling," in *SIGMOD*, 2021, pp. 2023–2035.

[7] L. Wang, R. Christensen, F. Li, and K. Yi, "Spatial online sampling and aggregation," *Proceedings of the VLDB*, vol. 9, no. 3, pp. 84–95, 2015.

[8] D. Amagata, Y. Arai, S. Fujita, and T. Hara, "Learned k-nn distance estimation," in *SIGSPATIAL*, 2022, pp. 1–4.

[9] K. Aoyama, D. Amagata, S. Fujita, and T. Hara, "Simpler is much faster: Fair and independent inner product search," in *SIGIR*, 2023, p. 2379–2383.

[10] S. Har-Peled and S. Mahabadi, "Near neighbor: Who is the fairest of them all?" *NeurIPS*, vol. 32, 2019.

[11] M. Aumüller, S. Har-Peled, S. Mahabadi, R. Pagh, and F. Silvestri, "Sampling a near neighbor in high dimensions—who is the fairest of them all?" *ACM Transactions on Database Systems*, vol. 47, no. 1, pp. 1–40, 2022.

[12] M. Aumüller, R. Pagh, and F. Silvestri, "Fair near neighbor search: Independent range sampling in high dimensions," in *PODS*, 2020, pp. 191–204.

[13] F. Olken and D. Rotem, "Random sampling from databases: A survey," *Statistics and Computing*, vol. 5, pp. 25–42, 1995.

[14] Y. Tao, "Algorithmic techniques for independent query sampling," in *PODS*, 2022, pp. 129–138.

[15] P. Afshani and J. M. Phillips, "Independent range sampling, revisited again," in *SoCG*, 2019.

[16] X. Hu, M. Qiao, and Y. Tao, "Independent range sampling," in *PODS*, 2014, pp. 246–255.

[17] F. Olken and D. Rotem, "Random sampling from b+ trees," in *VLDB*, 1989, pp. 269–277.

[18] P. Afshani and Z. Wei, "Independent range sampling, revisited," in *ESA*, 2017.

[19] J. Gan and Y. Tao, "Dynamic density based clustering," in *SIGMOD*, 2017, pp. 1493–1507.

[20] S. Rahul and R. Janardan, "A general technique for top-k geometric intersection query problems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 12, pp. 2859–2871, 2014.

[21] H. Edelsbrunner, *Dynamic Rectangle Intersection Searching*, 1980.

[22] F. Zhang, M. Jiang, and S. Wang, "Efficient dynamic weighted set sampling and its extension," *Proceedings of the VLDB*, vol. 17, no. 1, pp. 15–40, 2023.

[23] P. Bouros and N. Mamoulis, "A forward scan based plane sweep algorithm for parallel interval joins," *Proceedings of the VLDB*, vol. 10, no. 11, pp. 1346–1357, 2017.

[24] D. Piatov, S. Helmer, and A. Dignös, "An interval join optimized for modern hardware," in *ICDE*, 2016, pp. 1098–1109.

[25] P. Bouros, N. Mamoulis, D. Tsitsigkos, and M. Terrovitis, "In-memory interval joins," *The VLDB journal*, vol. 30, pp. 667–691, 2021.

[26] F. Cafagna and M. H. Böhlen, "Disjoint interval partitioning," *The VLDB Journal*, vol. 26, no. 3, pp. 447–466, 2017.

[27] D. Piatov, S. Helmer, A. Dignös, and F. Persia, "Cache-efficient sweeping-based interval joins for extended allen relation predicates," *The VLDB Journal*, vol. 30, no. 3, pp. 379–402, 2021.

[28] A. Dignös, M. H. Böhlen, and J. Gamper, "Overlap interval partition join," in *SIGMOD*, 2014, pp. 1459–1470.

[29] A. J. Walker, "New fast method for generating discrete random numbers with arbitrary frequency distributions," *Electronics Letters*, vol. 8, no. 10, pp. 127–128, 1974.

[30] M. S. Daskin and K. L. Maass, "The p-median problem," in *Location science*, 2015, pp. 21–45.

[31] P. Bouros, https://github.com/pbour/hint, 2022.

[32] https://www.kaggle.com/datasets/swaptr/bitcoin-historical-data.

[33] https://www.kaggle.com/datasets/thegurusteam/spanish-high-speed-rail-system-ticket-pricing.

[34] https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[35] M. Kaufmann, A. A. Manjili, P. Vagenas, P. M. Fischer, D. Kossmann, F. Färber, and N. May, "Timeline index: a unified data structure for processing queries on temporal data in sap hana," in *SIGMOD*, 2013, pp. 1173–1184.

[36] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "Sap hana database: Data management for modern business applications," *SIGMOD Record*, vol. 40, no. 4, pp. 45–51, 2012.

[37] d. B. Mark, C. Otfried, v. K. Marc, and O. Mark, *Computational Geometry Algorithms and Applications*, 2008.

[38] P. K. Agarwal, L. Arge, and K. Yi, "An optimal dynamic interval stabbing-max data structure?" in *SODA*, 2005, pp. 803–812.

[39] J. Xu and H. Lu, "Efficiently answer top-k queries on typed intervals," *Information Systems*, vol. 71, pp. 164–181, 2017.

[40] P. Bouros, K. Lampropoulos, D. Tsitsigkos, N. Mamoulis, and M. Terrovitis, "Band joins for interval data," in *EDBT*, 2020, pp. 443–446.

[41] P. Bouros and N. Mamoulis, "Interval count semi-joins," in *EDBT*, 2018, pp. 425–428.

[42] M. Aumuller, S. Har-Peled, S. Mahabadi, R. Pagh, and F. Silvestri, "Fair near neighbor search via sampling," *SIGMOD Record*, vol. 50, no. 1, pp. 42–49, 2021.

[43] M. Aumüller, S. Har-Peled, S. Mahabadi, R. Pagh, and F. Silvestri, "Sampling near neighbors in search for fairness," *Communications of the ACM*, vol. 65, no. 8, pp. 83–90, 2022.