Relevance queries for interval data

Panagiotis Bouros
Institute of Computer Science
Johannes Gutenberg University Mainz, Germany
Germany
bouros@uni-mainz.de

Nikos Mamoulis
Department of Computer Science and Engineering
University of Ioannina, Greece
Greece
nikos@cse.uoi.gr

ABSTRACT

A wide range of applications manage large collections of interval data. For instance, temporal databases manage validity intervals of objects or versions thereof, while in probabilistic databases attribute values of records are associated with confidence or uncertainty intervals. The main search operation on interval data is the retrieval of data intervals that intersect (i.e., overlap with) a query interval (e.g., find records which were valid in September 2020, find temperature readings with non-zero probability to be within [24, 26] degrees). As query results could be many, we need mechanisms that filter or order them based on how relevant they are to the query interval. We define alternative relevance scores between a data and a query interval based on their (relative) overlap. We define relevance queries, which compute only a subset of the most relevant intervals that intersect a query. Then, we propose a framework for evaluating relevance queries that can be applied on popular domain-partitioning interval indices (interval tree and HINT). We present experiments on real datasets that demonstrate the efficiency of our framework over baseline approaches.

CCS CONCEPTS

• **Information systems** → **Unidimensional range search**; **Query operators**.

KEYWORDS

Interval data, range query, selection query, ranking query, top-k

ACM Reference Format:
Panagiotis Bouros and Nikos Mamoulis. 2025. Relevance queries for interval data. In *Proceedings of Proceedings of the 2022 International Conference on Management of Data (SIGMOD '25).* ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

A wide range of applications manage and search large collections of interval data, including temporal databases [7, 36], probabilistic databases [12, 17], anonymized databases [33], XML databases [24, 25], spatial databases [23], and data streaming applications [4]. The most popular query over interval collections retrieves the intervals that overlap with a query point or range. Formally, consider a collection S of intervals, such that each $s \in S$ is defined by a pair of values $[s.start, s.end]$, where $start \leq s.end$.[1] Given a *query interval* $q = [q.start, q.end]$, the objective of the *range overlap* query is to find the subset S^q of S, such that for every $s \in S^q$, the intersection $s \cap q = [\max\{q.start, s.start\}, \min\{q.end, s.end\}]$ is non-empty. Below, we list important applications of interval search:

Temporal databases. In temporal databases [7, 36], records or versions thereof are associated with validity time intervals based on transaction time, valid time, or both. *Pure-timeslice* queries [32] ask for records valid sometime in a query time interval; these are equivalent to range-overlap queries. *Range-timeslice* queries ask for records valid in a query interval, also having their key (or some other attribute) in a given range. These two types of *temporal selections* are included in SQL extensions [26, 29] and implemented in PostgreSQL[2], Oracle Workspace Manager, IBM DB2 [34], Microsoft SQL Server[3], Teradata [1], and MariaDB[4].

Uncertain databases. Due to resource limitations or errors in sensing, an uncertainty interval is often used to model the possible range of actual values of an object [12]. For instance, sensors report their readings (e.g., temperature) periodically or only when they deviate a lot from the previously transmitted value. The price of a stock within a certain time bound (a few seconds) is approximated by a $[min, max]$ interval [2]. The actual value of the object is assumed to be within the uncertainty interval and its probability outside the interval is 0. Given a collection of such uncertainty intervals, a *probabilistic range query* is also expressed by an interval (data range) and the objective is to retrieve the set of all tuples (s_i, p_i), where s_i is the uncertainty interval of the reading and p_i is the non-zero probability that the actual value of s_i is inside the query range. In effect, this is a range overlap query.

XML document encodings. Finding the relationship (e.g., ancestor, sibling, etc.) between nodes in tree representations of XML documents facilitates the evaluation of XPath queries. Grust [24] encodes each node v by a $[pre(v), post(v)]$ interval, where $pre(v)$ and $post(v)$ are the ranks of v in the preorder and postorder traversal of the tree, respectively. Retrieving the nodes relative to a query node v is then translated to interval search operations (e.g., interval containment search retrieves the descendants).

Anonymized data. Generalization is one of the most popular anonymization approaches for relational data [39]. The values of a sensitive column are clustered and each original value is replaced by

SIGMOD '25, June 22–27, 2025, Berlin, Germany

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

[1] For the ease of exposition, we consider intervals that are closed at both ends. The techniques that we propose in this work can also directly be applied on collections of intervals that may be open at one or both ends.
[2] http://wiki.postgresql.org/wiki/Temporal_Extensions
[3] http://learn.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables
[4] http://mariadb.com/kb/en/system-versioned-tables/

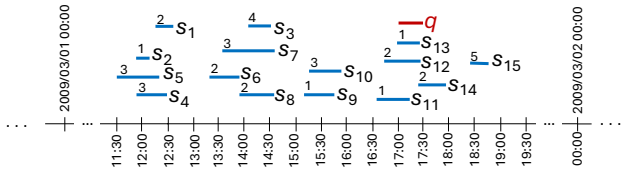

Figure 1: Example of data intervals and a query

a range which includes all values in the cluster where the value belongs. Then, range queries on the original column become interval overlap queries on the anonymized column, similar to probabilistic range queries over uncertain data [12]. PostgreSQL v. 9.2 includes native ranged data types, supported by range search operators.[5]

Managing intervals for the efficient evaluation of range overlap queries is a well-studied problem in the past 40 years for both memory-resident and disk-based data. A classic, worst-case optimal, data structure for intervals is the *interval tree* [22], originally used for the support of windowing operations in computational geometry [18]. Kriegel et al. adapted it for storage and search of intervals in a relational database [28]. The state-of-the-art data structure for intervals in main memory is HINT [14, 15]. To support range-timeslice queries in temporal databases, a number of composite indices have been proposed, most notably the multi-version B-tree [5, 30], the timeline index [27] (implemented in SAP HANA), and the recently proposed LIT [16]. These methods primarily index the records by time and secondarily by other attributes (such as record keys), as (typically) the time predicate is more selective.

Figure 1 shows a sample of intervals from the TAXIS dataset[6], which we use in our experiments. The dataset includes 169M NYC taxi trip records in 2009; the figure shows a tiny sample of the trips timespan on March 1, 2009 as intervals s_1 to s_{15}. The numbers on top of each interval indicate the number of passengers on that trip. Overall, 327K taxi trips took place on March 1, 2009 and the number of passengers in each trip ranged from 1 to 5. For each trip, the pick-up and drop-off locations are recorded but not the route in-between. Assume that a suspect involved in a crime was eye-witnessed in a cab between 17:00 and 17:30, denoted by interval q in the figure. In the example, taxi trips denoted by s_{11} to s_{14} overlap with q, so these trips should be investigated (i.e., by finding the license plates of the corresponding cabs and questioning the drivers). This corresponds to a pure-timeslice query [32]. The number of taxi trips that overlap with the query interval 17:00 to 17:30 on March 1st, 2009 is 12432. The cost to retrieve them using a HINT index [14] built on the entire data collection is 10 nsec, while a naive approach that accesses all intervals to find those that overlap with q is 65 msec. If additional information is known about the suspect, i.e., he was alone in the cab, the query would retrieve only s_{11} and s_{13} in Figure 1, i.e., trips with 1 passenger only. This range-timeslice query on the entire TAXIS dataset returns 8733 results. The cost to retrieve these results using a HINT index on all data and post-filtering by the passenger number is 50 nsec, while the cost of accessing only the single-passenger trips and verifying q on them is 42 msec. This example indicates that interval indexing such as the interval tree and HINT, are useful even if there are additional filters on other non-interval attributes.[7] Composite indices, such as LIT (which uses HINT to index part of the data), do even better, as shown in [16].

Motivation. In the previous example, regardless whether there is an additional selection on the number of passengers or not, the interval overlap query (17:00 to 17:30 on March 1st, 2009) returns numerous results. Examining these results one-by-one (e.g., questioning the taxi drivers that drove these suspect trips and following up at the drop-off points) is tedious and resource-consuming. It would make sense to either restrict the results to those that have large relative overlap with the query interval, or rank them based on their overlap-based similarity to the query. For example, we may want to examine the trips that overlap at least 80% of the query interval, or we may want to retrieve the ones that have the highest probability to overlap with the query.

Contributions. We first define the concept of the *relevance* for a data interval s to a query q, denoted by $Rel(s, q)$. We provide alternative definitions that could be useful in different application scenaria; i.e., based on the absolute length $|s \cap q|$ of the intersection, based on the data- or query-relative length ($|s \cap q|/|s|$ or $|s \cap q|/|q|$), or based on the symmetric relative length ($|s \cap q|/|s \cup q|$). In probabilistic databases, $|s \cap q|/|s|$ computes the probability that an uncertain value in range s is in the query interval q. Absolute ($|s \cap q|$) and query-relative ($|s \cap q|/|q|$) relevance is meaningful in applications with temporal data. For example, assume that q is a time period (e.g., a month) during which a rumor spread. The objective could be to find social network users who had been propagating the rumor for the most time within q. Symmetric relevance ($|s \cap q|/|s \cup q|$) is a continuous-space analog of Jaccard similarity and can be used to detect the most similar intervals to the query interval q; e.g., find web sessions whose timespan was very similar to the timespan of a fraudulent activity, to identify suspects. Keyword web queries (e.g., "war" and "peace") can be semantically related if the time periods when they are popular are similar [10]. Finally, as shown in [31], temporal queries may have too many or too few results, so defining approximate matches and ranking them is more practical.

We study two types of relevance queries for interval data. *Threshold-based* queries take as input a threshold θ and select only the data intervals s with relevance $Rel(s, q)$ at least θ; *ranking* queries take as input a positive integer k and return the top-k data intervals having the highest relevance to q. For example, in Figure 1, the taxi trips with the highest relevance to q using $|s \cap q|/|q|$ are s_{12}, s_{13}; if we limit search to trips with only one passenger, the most relevant trip is s_{13}. If relevance is defined as $|s \cap q|/|s|$, then the most relevant trip is s_{13} regardless whether there is a filter that the number of passengers is 1. If q is the uncertain time interval where the suspect was seen, and we look for the trips that overlap q with $\theta = 80\%$ probability, then we obtain s_{12}, s_{13} as results (or just s_{13} if we are looking for single-passenger taxi trips). The user may set appropriate values for θ and k depending on the application; in our taxi suspect example, we could set a small value, e.g., $k = 10$, since following up with the results can be resource demanding. Implementations of uncertain

[5] https://wiki.postgresql.org/images/7/73/Range-types-pgopen-2012.pdf
[6] http://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[7] Using an interval or another index for a composite query is a query optimization problem. Based on the analysis in [14], we can estimate the selectivity and the cost of an interval overlap query, enabling the optimizer to consider predicates on intervals.

data management systems [13, 35] allow for the specification of probabilistic thresholds in range-overlap queries.

A baseline approach to evaluate relevance queries is to first find all data intervals that overlap with q and then for each such object compute its relevance and output only the qualifying ones (based on θ or k). To this end, any existing interval index can be utilized. However, this method is expected to be slow, especially if the size of S^q is large. We propose a more efficient, general framework for relevance queries that can be used with any partition-based interval index, including the interval tree [22] and HINT [14]. For each partition P of the interval index, our approach requires knowledge about the minimum and maximum *start* and *end* points of the intervals assigned to P. In most cases this information can be readily available from the index, since intervals are typically sorted there. In case they are not, such information can be computed and maintained at minimal cost, as we will discuss. Given these statistics and q, we show how to derive fast upper and lower relevance bounds for the intervals in each partition. These bounds make it possible to prune partitions or obtain results in them without computations; they can also be used to define an access order for the partitions in ranking queries.

Our contributions can be summarized as follows:

- This is the first work, to our knowledge, that defines and studies relevance queries over interval data, based on alternative definitions, to be used by various applications that manage large volumes of interval data.
- We propose a *unified* approach for the efficient evaluation of relevance queries that applies on any off-the-shelf interval index. Our framework only requires statistics about the minimum and maximum end points of the intervals in each partition. Our method computes provable upper and lower bound relevance scores for each partition in $O(1)$ time, and uses them to prune partitions or schedule their access.
- We conduct an experimental study on real datasets demonstrating that our framework, when applied using state-of-the-art interval indices (interval tree, HINT) can reduce the relevance query processing cost by orders of magnitude.

Outline. The rest of the text is organized as follows. Section 2 defines the problem under study and includes the necessary background in indexing intervals. Section 3 presents our methodology for efficient evaluation of relevance queries over popular interval indices. Section 4 presents our experimental analysis. Section 5 discusses the related work. Finally, Section 6 concludes the paper.

2 PRELIMINARIES

We first introduce necessary notation and formally define the problem of relevance search on interval data. Then, we briefly describe two state-of-the-art indexing structures for interval data; later in the paper, we will elaborate on how to evaluate relevance queries efficiently using these structures.

2.1 Notation and problem definitions

Given a discrete or continuous 1D space, an interval is defined by a starting and an ending point in this domain. For instance, in the space of all non-negative integers $\mathbb{N}$, an interval $[start, end]$ with $start, end \in \mathbb{N}$ and $start \leq end$, is the subset of $\mathbb{N}$, which includes all integers x with $start \leq x \leq end$.

We denote by $[s.start, s.end]$ the interval which is associated to a data object s; for example, s could be a version of a record in a temporal database, or a probabilistic object in an uncertain database. We denote by $|s|$ the extent (i.e., length) of an object s, which equals the extent of its associated interval, i.e., $|s| = s.end - s.start$ in the domain is continuous or $|s| = s.end - s.start + 1$ if the domain is discrete. Given a query interval $q = [q.start, q.end]$, the intersection of an object s with q is $s \cap q = [\max\{s.start, q.start\}, \min\{s.end, q.end\}]$. Last, we denote by $Rel(s, q)$ the relevance of an object s to a query q. We introduce four alternative definitions for $Rel(s, q)$:

$$Rel_a(s,q) = |s \cap q| \quad (1) \qquad Rel_{rd}(s,q) = \frac{|s \cap q|}{|s|} \quad (3)$$

$$Rel_r(s,q) = \frac{|s \cap q|}{|s \cup q|} \quad (2) \qquad Rel_{rq}(s,q) = \frac{|s \cap q|}{|q|} \quad (4)$$

where $s \cup q$ denotes the interval that covers the collective range of s, q, i.e., $s \cup q = [\min\{s.start, q.start\}, \max\{s.end, q.end\}]$. Equation 1 computes an *absolute* relevance of s to q, while Equations 2–4, a *relative* relevance. For the relative relevance, we particularly distinguish between a *symmetric* version defined in Equation 2 and the *data*- or *query*-relative *asymmetric* one, defined in Equations 3 and 4, respectively.

Given a collection of data objects S and a query interval q, we next define two variants of relevance search on S.

Definition 2.1 (Threshold-based). Let θ be a relevance threshold; the *threshold-based relevance* query denoted by $\theta RelQuery(S, q)$, returns all objects in S whose relevance to query interval q exceeds θ. Formally,

$$\theta RelQuery(S, q) = \{s \in S : Rel(s, q) \geq \theta\}$$

Definition 2.2 (Ranking). Let k a positive integer; the *ranking relevance* query denoted by $kRelQuery(S, q)$, returns the k subset of the objects in S with the highest relevance to query interval q, Formally,

- $|kRelQuery(S, q)| = k$, and
- $kRelQuery(S, q) = \{s \in S : \forall s' \in S \setminus kRelQuery(S, q), Rel(s', q) \leq Rel(s, q)\}$

2.2 Background on indexing intervals

We revisit the popular interval tree [22] and the recently proposed HINT [14, 16] for indexing a collection of data objects $\mathcal{S}$. Based on the analysis in [16], the two structures represent the best profiles in terms of query performance and storage requirements. Specifically, the interval tree typically has the lowest space requirements while HINT achieves the highest query throughput.

2.2.1 Interval tree. The interval tree [22] is a binary search tree, which occupies $O(n)$ space and answers selection queries on interval data in $O(\log n + K)$ time, where K is the number of query results. The tree divides the 1D domain hierarchically as follows. We first use the median M_1 of the $2n$ endpoints of all intervals to define the root v_1 of the tree and partition the set S of data intervals into three sets: the intervals which include M_1 are placed in the

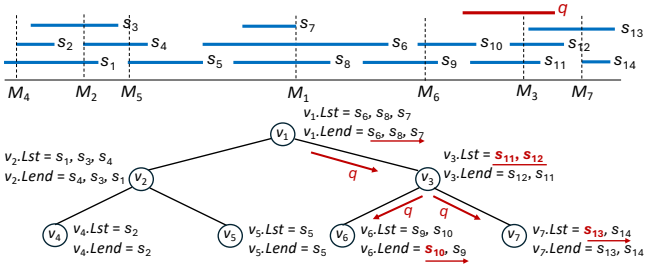

Figure 2: Example of an interval tree

ALGORITHM 1: Range query on the interval tree

Input : Interval tree $\mathcal{I}$, query interval q
Output : set of all intervals that overlap with q

```
1  Function Search(node v, query interval q):
2      if v.M < q.start then
3          s ← v.Lend.first();
4          while s.end ≤ q.start do
5              output s;
6              s ← v.Lend.next();
7          Search(v.rightchild, q);
8      else if v.M > q.end then
9          s ← v.Lst.first();
10         while s.start ≥ q.end do
11             output s;
12             s ← v.Lst.next();
13         Search(v.leftchild, q);
14     else                                   ▷ q.start ≤ M ≤ q.end
15         output all s ∈ v.Lst;
16         Search(v.leftchild, q);
17         Search(v.rightchild, q);
18 Search(I.root, q);                  ▷ Traverse the tree, depth-first
```

root, the set of intervals S_L which end before M_1 are assigned to the left subtree of the root and the set of intervals S_R which begin after M_1 are assigned to the right subtree of the root. The intervals in the root v_1 are placed in two sorted lists: $v_1.Lst$ keeps the intervals in increasing order of their $s.start$ endpoint and $v_1.Lend$ keeps the intervals in decreasing order of their $s.end$ endpoint. S_L and S_R are divided recursively using their respective medians M_2 and M_3, to define v_2 and v_3, the left and right children of v_1. Figure 2 shows an example of an interval tree (bottom) for a set of 14 intervals (top).

Point and range queries can be evaluated as shown by Algorithm 1. Consider the range query $q = [q.start, q.end]$ shown in Figure 2. Since $q.start > M_1$, we scan $v_1.Lend$ to find any intervals in the root that overlap with q. As soon as we access one interval (e.g., s_6) for which the end point is smaller than $q.start$, we terminate the scan (e.g., $s_6.end < q.start$) because subsequent intervals s (e.g., s_8, s_7) have an even smaller $s.end$. Condition $q.start > M_1$ dictates that we have to search the right subtree, so we access v_3 and compare M_3 to q. In this case, M_3 is included in q, so all intervals assigned to v_3 are guaranteed to overlap with q. To report them, we can scan any of $v_3.Lst$ or $v_3.Lend$. Since M_3 is included in q, we have to recursively search *both* subtrees of v_3. To the left, $M_6 < q.start$, so we scan $v_6.Lend$, obtain s_{10}, and stop the scan at s_9 since $s_9.end < q.start$. To the right, $M_7 > q.start$, so we scan $v_7.Lst$ and obtain s_{13} (scanning is stopped at s_{14} since $s_{14}.start > q.end$).

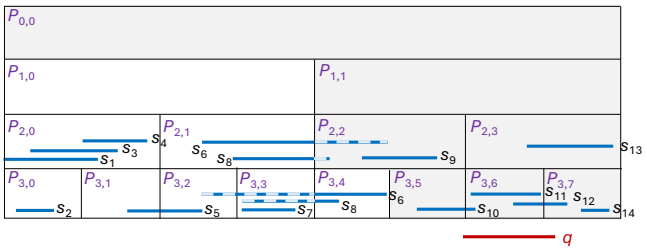

Figure 3: Example of HINT

2.2.2 HINT. HINT [14, 16] hierarchically and uniformly divides the domain into 2^ℓ partitions for $\ell = 0$ to m, defining $m + 1$ levels, as shown in Figure 3. Partitions at level ℓ are denoted by $P_{\ell,0}$ to $P_{\ell,2^\ell-1}$. Each interval s is normalized, discretized in the $[0, 2^{m-1}]$ domain, and assigned to the smallest set of partitions from all levels that cover s (at most 2 partitions per level). For example, in Figure 3, intervals s_1, s_3, s_4 are assigned to partition $P_{2,0}$ only, interval s_5 to partitions $P_{3,1}$ and $P_{3,2}$, and intervals s_6, s_8 to partitions $P_{2,1}$ and $P_{3,4}$. The intervals in each partition P are split into two divisions (sub-partitions): those that start *inside* P (called *originals*), denoted by P^O, and those that start *before* P (called *replicas*), denoted by P^R. For example, intervals s_6, s_8 are in $P^O_{2,1}$ and in $P^R_{3,4}$.

Given a selection query $q = [q.start, q.end]$, at each index level ℓ, only the sequence of partitions $P_{\ell,i}$ that overlap with q are accessed. For example, for query q in Figure 3, the accessed partitions are gray-shaded. To avoid producing duplicate results and save on unnecessary accesses and comparisons, originals and replicas divisions are only processed in the first accessed partition at each level ℓ, while for the remaining partitions only originals are processed. Hence, for query q in Figure 3, in partition $P_{3,5}$, we access $P^O_{3,5}$, containing intervals than begin inside $P_{3,5}$, including s_{10}, and $P^R_{3,5}$ which includes any intervals that begin before $P_{3,5}$ and end inside it. In $P_{3,6}$, we only access $P^O_{3,6}$ which includes s_{11} and s_{12}, but do not access $P^R_{3,6} = \{s_{10}\}$, thus avoiding considering s_{10} again.

At each level ℓ, the partitions that overlap a query q span from $P_{\ell,f}$ to $P_{\ell,l}$, where f and l are the ℓ-bit prefixes of $q.start$ and $q.end$, respectively. For all partitions $P_{\ell,i}$ with $f < i < l$, we can simply report all intervals in $P^O_{\ell,i}$ (no comparisons are required). To minimize the required comparisons for $P_{\ell,f}$ and $P_{\ell,l}$, the search algorithm accesses HINT bottom-up, i.e., from level m to level 0. If $P_{\ell,f}$ (resp. $P_{\ell,l}$) starts (resp. ends) at the same point as $P_{\ell+1,f}$ (resp. $P_{\ell+1,l}$), then no comparisons are required for $P_{\ell,f}$ (resp. $P_{\ell,l}$) and for all $P_{\ell',f}$ (resp. $P_{\ell',l}$), $\ell' < \ell$. Given this, the search algorithm conducts comparisons at just four partitions by expectation [14]. The number of comparisons can be further reduced by splitting the P^O division (i.e., originals) of a partition P into *subdivisions* $P^{O_{in}}$ and $P^{O_{aft}}$, so that $P^{O_{in}}$ ($P^{O_{aft}}$) holds the intervals from P^O that *end* inside (resp. after) P. Similarly, each P^R is divided into $P^{R_{in}}$ and $P^{R_{aft}}$. For example, in partition $P_{3,6}$, intervals s_{11} and s_{12} are both "original" because they begin in the subdomain defined by $P_{3,6}$, so they are placed in $P^O_{3,6}$. Division $P^O_{3,6}$ is further subdivided into $P^{O_{in}}_{3,6}$ which includes s_{11} and $P^{O_{aft}}_{3,6}$ which includes s_{12}. Algorithm 2 illustrates the pseudocode for computing range queries with HINT.[8]

[8] For simplicity, we omitted the special case when the first and the last relevant partitions coincide, i.e., $f = l$.

ALGORITHM 2: Range query on HINT

Input : HINT index $\mathcal{H}$, query interval q
Output : set of all intervals that overlap with q

1 **foreach** *level* $\ell = m$ **to** 0 **do** ▷ traverse index, bottom-up
2 $f \leftarrow prefix(\ell, q.start)$; ▷ first overlapping partition
3 $l \leftarrow prefix(\ell, q.end)$; ▷ last overlapping partition
4 **output** all $s \in P^{O_{in}}_{\ell,f}$ with $s.start \leq q.end$;
5 **output** all $s \in P^{O_{aft}}_{\ell,f}$;
6 **output** all $s \in P^{R_{in}}_{\ell,f}$ with $q.start \leq s.end$;
7 **output** all $s \in P^{R_{aft}}_{\ell,f}$;
8 **foreach** *partition* i *with* $f < i < l$ **do**
9 **output** all $s \in P^{O_{in}}_{\ell,i} \bigcup P^{O_{aft}}_{\ell,i}$;
10 **output** all $s \in P^{O_{in}}_{\ell,l} \bigcup P^{O_{aft}}_{\ell,l}$ with $s.start \leq q.end$;

Observe how the subdivisions of P^O and P^R are processed in a different fashion allowing to reduce the necessary comparisons; e.g., for the intervals in $P^{R_{in}}$ only their end is checked while for the intervals in $P^{R_{aft}}$ no checks are required as they all overlap with the query q, by definition. Lastly, to further accelerate Lines 4, 6 and 10 where interval endpoints are compared against the query interval q, the contents of $P^{O_{in}}$ and $P^{O_{aft}}$ are sorted by $s.start$ and of $P^{R_{in}}$, by $s.end$, similar to the interval tree lists Lst and $Lend$.

3 METHODOLOGY

The naive approach of processing relevance queries (Definitions 2.1 and 2.2) is to use an index, such as the interval tree or HINT (described in Section 2.2), to retrieve *all* data intervals that overlap with the query interval q. For every such interval s, we can compute $Rel(s, q)$ and either verify $Rel(s, q) \geq \theta$ (Def. 2.1) or keep track (in a heap) of the k intervals with the highest $Rel(s, q)$ (Def. 2.2). However, the naive approach may access more data than necessary. In this section, we present our methodology which prunes partitions that cannot produce results and reduces computations in accessed partitions as much as possible.

3.1 Overview

Figure 4 illustrates our framework and the steps of query evaluation. In a preprocessing (offline) phase, i.e., before any query arrives, for each partition (e.g., interval tree node or HINT partition), we compute the minimum and maximum start and end points of any interval in the partition. As we discuss in Section 3.2, these statistics can be readily available in the partitions.

Then, for a *threshold query* q (Definition 2.1), where we search for data intervals s having $Rel(s, q) \geq \theta$, we use the index to identify the partitions that may include intervals that overlap with q; for each such partition P, we use the minimum and maximum statistics for the start/end of intervals in P to compute $UB(P)$ and $LB(P)$, i.e., the upper and lower bound of $Rel(s, q)$ for any $s \in P$, respectively, in $O(1)$ time, as explained in Section 3.3. If $UB(P) < \theta$ holds, we can prune the entire partition, without accessing any data in it. If $LB(P) \geq \theta$, we report all data intervals in P as results without having to compute their intersection with q and to conduct any comparisons. If neither of the above holds, then we have $LB(P) < \theta \leq UB(P)$ and we do have to access the partition and for each $s \in P$ that intersects q compute $Rel(s, q)$ and check if $Rel(s, q) \geq \theta$.

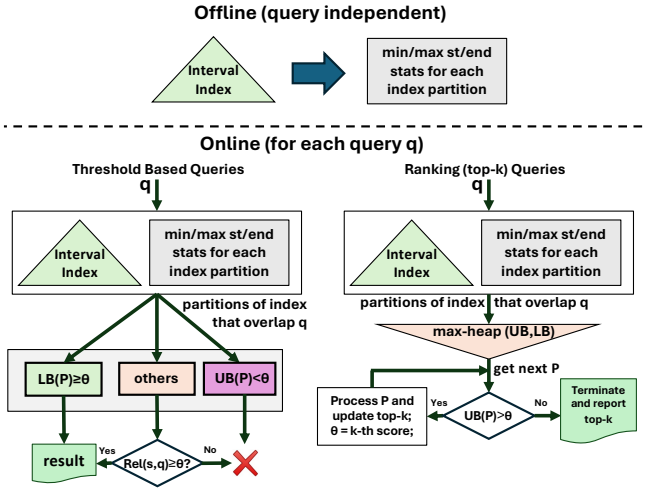

Figure 4: Proposed relevance framework

For a *ranking query* q (Definition 2.2), we use again the index to identify all partitions with intervals that may overlap with q, but do not access each of them immediately. Instead, we add all these partitions P to a max-heap, to be accessed in decreasing order of their $UB(P)$, i.e., the upper bound of $Rel(s, q)$ for any $s \in P$; we break ties by $LB(P)$. Then, we start processing the partitions in priority order, access each data interval s in each partition, compute their relevance score $Rel(s, q)$ to q, and keep track of the k intervals with the largest $Rel(s, q)$ in a min-priority query $\mathcal{Q}$. If the next partition P to process has $UB(P) \leq \theta$, where θ is the smallest $Rel(s, q)$ in $\mathcal{Q}$, we stop, reporting $\mathcal{Q}$ as the top-k results.

3.2 Min/Max statistics in partitions

For each partition, our framework needs four numbers: the minimum and maximum *start* and the minimum and maximum *end* of the intervals in the partition. For example, for partition $P^{O_{in}}_{2,0}$ in Figure 3, we have $minst = s_1.start$, $maxst = s_4.start$ and $minend = s_1.end$, $maxend = s_4.end$. In principle, these bounds can be obtained from respective statistics which we can maintain together with the index. They take up to $O(1)$ space per partition, so their space requirements are bounded by the number of non-empty partitions.

For the interval tree, there is no space or maintenance overhead for the min/max statistics. Recall that in each node v, the contained intervals are kept in two lists: $v.Lst$ and $v.Lend$, sorted by their *start* and by their *end* point, respectively. Hence, $v.minst$, $v.maxst$ and $v.minend$, $v.maxend$ equal the first and the last entry in $v.Lst$ and $v.Lend$, respectively. When a new interval is inserted to v or an existing interval is deleted from v, $v.Lst$ and $v.Lend$ are updated and the order in each list is maintained, so the min/max statistics in v are still obtained from the first and last entries in the lists.

In HINT, for a partition P, we distinguish between two cases depending on whether the contents of the $P^{O_{in}}$, $P^{O_{aft}}$ and $P^{R_{in}}$ subdivisions are sorted or not (see Section 2.2.2). If no sorting is imposed, we store and maintain all four *minst*, *maxst* and *minend*, *maxend* statistics for every subdivision. When a new interval s is added to a subdivision, it suffices to compare $s.start$ and $s.end$

against these four numbers and update the statistics if needed. When an existing interval is deleted, we need to scan the contents of the subdivision to update the statistics, only if $s.start = minst$ or $s.start = maxst$ and $s.end = minend$ or $s.end = maxend$. Since deletion requires scanning the subdivision to find s and delete it, asymptotically, updating the statistics does not impose an overhead to the update process. On the other hand, when sorting is used, we explicitly store and update only *minend*, *maxend* for the $P^{O_{in}}$, $P^{O_{aft}}$ subdivisions, which are sorted by the *start* point, and *minst*, *maxst* for $P^{R_{in}}$ which is sorted by *end*, similar to the interval tree and lists *Lin* and *Lend*, respectively. Note that $P^{R_{aft}}$ employs no sorting and so, all four bounds are stored and maintained. Overall, for HINT, (1) the time complexity of updates is not affected by the maintenance of min/max statistics and (2) we need $O(1)$ space per non-empty subdivision to store its min/max statistics.

3.3 Computing relevance bounds

Given a partition P of an interval index, a query interval q, and a definition *Rel* for relevance (e.g., Eq. 3), our framework computes an upper bound $UB(P)$ and a lower bound $LB(P)$ of $Rel(s, q)$ for any data interval $s \in P$, to be used for potentially pruning or ranking partitions.

3.3.1 Upper relevance bounds. Our first theoretical result is that, in all definitions of relevance, the best possible interval in a partition is the *shortest possible interval* in the partition that *maximizes* the *absolute overlap* with the query interval. This interval can be derived from the minimum and maximum statistics (see Section 3.2) and the query interval. More formally:

Theorem 3.1. *Let minst and maxst (resp. minend and maxend) be the smallest and largest start (resp. end) points of any interval in a partition P. In addition, let q be a query interval. The data interval s_{ub} in P that gives the largest possible relevance score for any definition of relevance $Rel(s, q)$ has $s_{ub}.start = \min\{\max(q.start, minst), maxst\}$ and $s_{ub}.end = \max\{\min(q.end, maxend), minend\}$.*

Proof. (Sketch.) Provided that we do not know the contents of P, but only its min/max statistics (i.e., *minst*, *maxst*, *minend*, and *maxend*), we will first prove that the shortest interval in P that maximizes the overlap with q is s_{ub}. Based on the statistics, $s_{ub}.start$ ranges within $[minst, maxst]$. If $q.start < minst$, then $s_{ub}.start$ should be *minst* because any value greater than *minst* will result in smaller overlap, regardless where $q.end$ lies. If $minst \leq q.start \leq maxst$, to maximize overlap $s_{ub}.start$ should be smaller than or equal to $q.start$, so, the shortest interval that maximizes the overlap with q should start at $q.start$. Finally, if $q.start \leq maxst$, $s_{ub}.start$ should be *maxst*, as setting it to a smaller value does not increase the overlap. The three cases, exemplified in Figure 5, can be combined to $s_{ub}.start = \min\{\max(q.start, minst), maxst\}$. $s_{ub}.end = \max\{\min(q.end, maxend), minend\}$ can be proved symmetrically. Overall, s_{ub} is the shortest possible interval $s \in P$ having maximal intersection $|s \cap q|$ with q.

Now, we will show, for all definitions of relevance $Rel(s, q)$, that if we change the endpoints of s_{ub} within their allowable bounds $Rel(s, q)$ cannot increase. For $Rel_a(s, q) = |s \cap q|$ and $Rel_{rq}(s, q) = |s \cap q|/|q|$, the proof is straightforward, as we have already shown

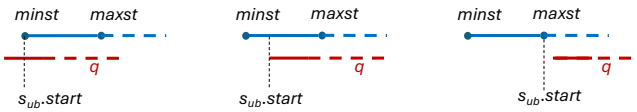

Figure 5: Three cases of $s_{ub}.start$

that s_{ub} is the interval that maximizes $|s \cap q|$. Hence, changing endpoints of s_{ub} can only decrease the relevance. Regarding $Rel_{rs}(s, q) = |s \cap q|/|s|$, reducing $s_{ub}.start$ or increasing $s_{ub}.end$ does not increase $|s \cap q|$, but increases $|s|$, so relevance decreases. Increasing $s_{ub}.start$ or decreasing $s_{ub}.end$ reduces the length of s, say by a, decreasing the denominator $|s|$ by a. The numerator $|s \cap q|$ can decrease by at most a. Hence, the fraction may only decrease as the numerator is smaller than or equal to the denominator and $x/y \geq (x - a)/(y - a)$ for all positive x, y, a, where $x \leq y$. Finally, for $Rel_r(s, q) = |s \cap q|/|s \cup q|$, reducing $s_{ub}.start$ or increasing $s_{ub}.end$ does not increase $|s \cap q|$, but can only increase $|s \cup q|$. Increasing $s_{ub}.start$ or decreasing $s_{ub}.end$ can reduce the numerator $|s \cap q|$ and the denominator $|s \cup q|$ by at most the same value. □

Theorem 3.1 is an important result, because it allows us to derive upper bounds for the partitions that overlap with the query q in an efficient and unified manner, regardless the type of the interval index and the relevance measure. We simply have to compute s_{ub} and use $Rel(s_{ub}, q)$ as the upper bound $UB(P)$.

3.3.2 Lower relevance bounds. Lower relevance bounds for the intervals in a partition P can be also derived from the minimum and maximum statistics in P. Similar to the s_{ub} for the upper relevance bound, the goal is to determine the data interval s_{lb} in P which minimizes the $|s_{lb} \cap q|$ overlap while maximizing $|s_{lb} \cup q|$ for Rel_r and $|s_{lb}|$ for Rel_{rq}. Under this premise, the process of determining s_{lb} heavily depends on the relevance definition and thus, we devise a case-based solution for $LB(P)$.

Theorem 3.2. *Let minst and maxst (resp. minend and maxend) be the smallest and largest start (resp. end) points of any interval in a partition P. Assume that $maxst \geq minend$. Also, let q be a query interval. The data interval s_{lb} in P that gives the lowest possible relevance score for $Rel_a(s, q)$ and $Rel_{rq}(s, q)$ has $s_{lb}.start = maxst$ and $s_{lb}.end = minend$.*

Proof. s_{lb} corresponds to the shortest possible valid interval in P. Also, s_{lb} is covered by all valid intervals in P. Hence, $|s_{lb} \cap q|$ is the smallest possible overlap between any interval in P and q. □

Following Theorem 3.2, we simply set $LB(P) = Rel_a(s_{lb}, q)$ or $LB(P) = Rel_{rq}(s_{lb}, q)$, depending on the relevance definition.

Note that if $maxst > minend$, s_{lb} is invalid as its start point is greater than its end point. In this case, $|s \cap q|$ in $LB(P)$ is the smallest possible interval length, if q fully covers $[minst, maxend]$ or set $LB(P) = 0$, otherwise. In both situations, $LB(P)$ corresponds to a very small (and hence useless lower bound), so for partitions where $maxst > minend$ (i.e., at the lowest level of HINT), it is not advisable to compute and use $LB(P)$ for Rel_a or Rel_{rq}.

For the relative $Rel_r(s, q)$ and the data-relative Rel_{rd} definitions, $LB(P)$ cannot be determined by a single s_{lb}, but by the four extreme cases of s_{lb} considering all four combinations of the *minst*, *maxst* and *minend*, *maxend* bounds.

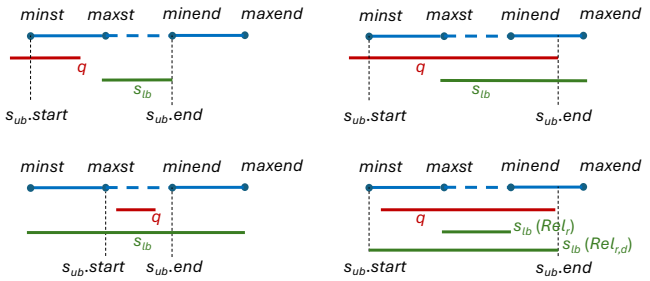

Figure 6: Four cases of lower bounds Rel_r and Rel_{rd}

Theorem 3.3. *Let minst and maxst (resp. minend and maxend) be the smallest and largest start (resp. end) points of any interval in a partition P. Assume that maxst ≥ minend. Also, let q be a query interval. The lower relevance bound for the relative $Rel_r(s,q)$ and the data-relative $Rel_{rd}(s,q)$ definitions is computed by:*

$$\begin{aligned} LB(P) = \min\{&Rel([minst, minend], q),\\ &Rel([minst, maxend], q),\\ &Rel([maxst, minend], q),\\ &Rel([maxst, maxend], q)\} \end{aligned}$$

Proof. (Sketch.) Assume s is any of $\{[minst, minend], [minst, maxend], [maxst, minend], [maxst, maxend]\}$. For each one of the two definitions of Rel, i.e., Rel_r or Rel_{rd}, we can prove that if we move $s.start$ toward $s_{ub}.start$ and/or move $s.end$ toward $s_{ub}.end$, where s_{ub} is defined by Theorem 3.1, $Rel(s,q)$ can only increase. (Details are case-based and they are omitted for brevity.) Hence, each of the four possible instances of s gives a local lower bound and the global lower bound is the lowest of the local ones. □

Finally, like before, the case where $maxst > minend$ should be treated specially. If q covers $[minst, maxend]$ the lower bound is 1 for Rel_{rd} or derived using the minimum possible interval length for Rel_r. If q does not cover $[minst, maxend]$, $LB(P) = 0$. Figure 6 exemplifies some cases of applying Theorem 3.3 ($maxst \geq minend$) to derive s_{lb}, i.e., the interval that gives the lower bound, for Rel_{rd} and Rel_r. Observe that, in both definitions of relevance, if we move the endpoints of s_{lb} towards the endpoints of the interval s_{ub} (that gives $UB(P)$), relevance cannot decrease.

3.4 Query processing algorithms

Finally, we show how to integrate the lower and upper relevance bounds of the index partitions in the computation of threshold-based and ranking relevance queries.

3.4.1 Threshold-based queries. We start off with $\theta RelQuery$. For the interval tree, we rely on the depth-first traversal depicted in Algorithm 1. Algorithm 3 shows parts of the modifications to Algorithm 1 for the $\theta RelQuery$, highlighted by gray shade. We detail the necessary modifications when $M < q.start$; similar changes are made when $M > q.end$ or $q.start \leq M \leq q.end$. For each visited node v, we first compute $UB(v)$ following the procedure outlined in Section 3.3.1. If $UB(v) \geq \theta$, we compute $LB(v)$, following Section 3.3.2 and if $LB(v) \geq \theta$ we directly report all contents of v (e.g., taken from $v.Lend$) without incurring any computations. Otherwise, depending on the value of M w.r.t. q, we traverse $v.Lst$ or

ALGORITHM 3: $\theta RelQuery$ on the interval tree

```
Input  : Interval tree I, query interval q, threshold θ
Output : all intervals that overlap with q and Rel(s, q) ≥ θ
1  Function Search(node v, query interval q, threshold θ)
2      if v.M < q.start then
3          if UB(v) ≥ θ then
4              if LB(v) ≥ θ then
5                  output all s ∈ v.Lend;
6              else
7                  s ← v.Lend.first();
8                  while s.end ≤ q.start do
9                      if Rel(s, q) ≥ θ then
10                         output s;
11                     s ← v.Lend.next();
12         Search(v.rightchild, q, θ);
13     ...                ▷ Similar changes to Lines 8-17 in Algorithm 1
14 Search(I.root, q, θ);          ▷ Traverse the tree, depth-first
```

ALGORITHM 4: $\theta RelQquery$ on HINT

```
Input  : HINT index H, query interval q
Output : set of all intervals that overlap with q
1  foreach level ℓ = m to 0 do          ▷ traverse index, bottom-up
2      f ← prefix(ℓ, q.start);          ▷ first overlapping partition
3      l ← prefix(ℓ, q.end);            ▷ last overlapping partition
4      if UB(P^Oin_{ℓ,f}) ≥ θ then
5          if LB(P^Oin_{ℓ,f}) ≥ θ then
6              output all s ∈ P^Oin_{ℓ,f};
7          else
8              output all s ∈ P^Oin_{ℓ,f} with s.start ≤ q.end and Rel(s, q) ≥ θ;
9      ...                ▷ Similar changes to Lines 5-10 in Algorithm 2
```

$v.Lend$ and conduct comparisons and computations of $Rel(s,q)$ for the intervals $s \in v$ that overlap with q. For example, consider the interval tree and the query q in Figure 2 and assume that we want to retrieve relevant intervals to q based on Rel_{rq} using $\theta = 0.3$. When accessing the root of the tree (v_1), we first compute the interval s_{ub} that gives the upper relevance bound of any interval in v_1 using $v_1.minst = s_6.start$, $v_1.maxend = s_6.end$. According to Theorem 3.1, $s_{ub} = [v_1.minst, v_1.maxend]$, so the upper relevance bound $UB(v_1)$ for v_1 is $Rel_{rq}(s_{ub}, q) = 0$. This means that we do not have to access any intervals in v_3 (i.e., the condition of Line 3 in Algorithm 3 is false). Then, we recursively search the right child of v_1 (Line 12) and for v_3, we compute $UB(v_3)$ and $LB(v_3)$, using Theorems 3.1 and 3.2, respectively, with $s_{ub} = [s_{11}.start, q.end]$ and $s_{lb} = [s_{12}.start, s_{11}.end]$. Both bounds are greater than $\theta = 0.3$, hence we know that all intervals in v_3 are query results (Line 5). The algorithm now searches recursively both children of v_3, i.e., v_6 and v_7. For v_6, $s_{ub} = s_{10}$, so $UB(v_6) < \theta$ and the contents of v_6 are not accessed. For v_7, $s_{ub} = [s_{13}.start, s_{14}.end]$ and $UB(v_7) > \theta$, but $s_{lb} = s_{14}$ and $LB(v_7) = 0$, hence, the intervals in v_7 are accessed to compute their relevance to q; from these s_{13} is reported as result.

For HINT, we accordingly adapt Algorithm 2 to include the additional checks for verifying overlapping intervals and for the relevance bounds. Algorithms 4 exemplifies these modifications

ALGORITHM 5: *kRelQuery* on the interval tree

```
Input  : Interval tree I, query interval q, number of results k
Output : k intervals that overlap with q, having the highest Rel
1  Function Search(node v, q, Q, k)
2      if v.M < q.start then
3          if UB(v) > Res(Q.top(), q) then
4              s ← v.Lend.first();
5              while s.end ≤ q.start do
6                  if Rel(s, q) > Res(Q.top(), q) then
7                      add s to Q;                          ▷ Update result
8                      if |Q| > k then
9                          remove Q.top();
10             s ← v.Lend.next();
11         Search(v.rightchild, q, Q, k);
12     ...                ▷ Similar changes to Lines 8–17 in Algorithm 1
13 initialize min-priority queue Q;                  ▷ top-k list for result
14 Search(I.root, q, Q, k);                ▷ Traverse the tree, depth-first
15 output Q;
```

ALGORITHM 6: *kRelQquery* on HINT

```
Input  : HINT index H, query interval q, number of results k
Output : k intervals that overlap with q, having the highest Rel
1  initialize min-priority queue Q;                  ▷ top-k list for result
2  foreach level ℓ = m to 0 do                  ▷ traverse index, bottom-up
3      f ← prefix(ℓ, q.start);                ▷ first overlapping partition
4      l ← prefix(ℓ, q.end);                   ▷ last overlapping partition
5      if UB(P^{Oin}_{ℓ,f}) > Res(Q.top(), q) then
6          foreach interval s ∈ P^{Oin}_{ℓ,f} with s.start ≤ q.end do
7              if Rel(s, q) > Res(Q.top(), q) then
8                  add s to Q;                              ▷ Update result
9                  if |Q| > k then
10                     remove Q.top();
11     ...                ▷ Similar changes to Lines 5–10 in Algorithm 2
12 output Q;
```

for Line 4 in Algorithm 2 and the $P^{O_{in}}_{\ell,f}$ subdivision of the first relevant partition on each level, again highlighted by gray shade. The subdivision is considered only if $UB(P^{O_{in}}_{\ell,f}) \geq \theta$ holds for its upper relevance bound. If $LB(P^{O_{in}}_{\ell,f}) \geq \theta$ also holds for the lower relevance bound then all contained intervals can be directly output; their overlap with the query is guaranteed. Otherwise (i.e., if $LB(P^{O_{in}}_{\ell,f}) < \theta$), we need to compute for every overlapping interval s (i.e., with $s.start \geq q.end$) its relevance score and output s only if $Rel(s, q) \geq \theta$. As an example consider query q and HINT in Figure 3 using Rel_{rq} and $\theta = 0.3$. Recall that the gray-shaded partitions (their subdivisions) in the figure are the ones to be accessed in a bottom-up fashion, since they overlap with the query range. From these partitions, subdivisions $P^{O_{aft}}_{3,5}$, $P^{O_{in}}_{3,7}$, $P^{O_{in}}_{2,2}$, $P^{R_{in}}_{2,2}$, are pruned (as per Line 4 of Algorithm 4) because their $UB(P)$ (derived from the corresponding intervals computed using Theorem 3.1) are smaller than θ. For subdivisions $P^{O_{in}}_{3,6}$, $P^{O_{aft}}_{3,6}$ and $P^{O_{in}}_{2,3}$, all contents are reported as results (as per Lines 5–6 of Algorithm 4), because their lower bounds (computed with the help of Theorem 3.2) exceed θ.

3.4.2 Ranking queries. We next discuss *kRelQuery*. For the interval tree, we can still capitalize on the depth-first approach of Algorithm 1, extended to filter out nodes using the upper relevance bound $UB(P)$; lower bounds are not useful in this context. The key idea is to maintain the k intervals with the highest relevance scores seen so far inside a min-priority queue Q, which will eventually contain the final result. Under this premise, the relevance $Res(Q.top(), q)$ of the top element in Q (i.e., the lowest relevance in Q) can be used for pruning similar to threshold θ for θ*RelQuery*. Algorithm 5 presents the pseudocode of this approach; for the interest of space, we again only discuss the case of $M < q.start$. In between Lines 3–10, the algorithm scans the $v.Lend$ list of the current node v only if its upper relevance bound exceeds the lowest relevance score in the current result, i.e., if $UB(v) > Res(Q.top(), q)$. If so, similar to Algorithm 3, we first identify each overlapping interval s to q in $v.Lend$ and then check whether this interval can be part of the result, i.e., if $Rel(s, q) > Rel(Q.top(), q)$ holds. Note that Lines 8–9 guarantee that queue Q can never contain more than k intervals. As an example, assume that we apply Algorithm 5 on Figure 2 for $k = 2$, using query q and Rel_{rq}. Node v_1 does not have any intervals that overlap with q; after accessing v_3, both its intervals s_{11} and s_{12} are added to $Q = \{s_{11}, s_{12}\}$ as currently the $k = 2$ most similar ones to q, with $Q.top() = s_{12}$. Upon reaching v_6, we compute $s_{ub} = s_{10}$ and find that $UB(v_6) < Rel_{rq}(s_{12}, q)$, so, the contents of v_6 need not be accessed (Line 5 of Algorithm 5). When accessing v_7, we also see that $UB(v_7) < Rel_{rq}(s_{12}, q)$, so there is no need to access the intervals in v_7 and the algorithm terminates reporting $Q = \{s_{11}, s_{12}\}$.

In a similar fashion, we modify Algorithm 2 to compute *kRelQuery* in a bottom-up fashion on HINT. Algorithm 6 illustrates part of the pseudocode, which covers again the case of the $P^{O_{in}}_{\ell,f}$ subdivision for the first relevant partition on every level ℓ. Similar to Algorithm 5, the subdivision is scanned only if $UB(P^{O_{in}}_{\ell,f}) > Rel(Q.top(), q)$. Then, each contained overlapping interval s with $Rel(s, q) > Rel(Q.top(), q)$ is used to update current result in Q. To illustrate Algorithm 6, consider the index and query q in Figure 3 and $k = 2$, using Rel_{rq}. The partitions are accessed bottom-up, and left-to-right for each level. Upon accessing $P_{3,5}$ from Level 3, Q is populated by s_{10}. Then, after accessing $P_{3,6}$, Q is updated to include s_{11} and s_{12}, which have higher relative overlap to q compared to s_{10}. Now, $Q.top() = s_{12}$. Partition $P_{3,7}$ is eliminated because $UB(P^{O_{in}}_{3,7}) = 0$. At level 2, partitions $P_{2,2}$ and $P_{2,3}$ are eliminated because their upper bounds are smaller than $Rel_{rq}(s_{12}, q)$ and the algorithm terminates reporting $Q = \{s_{11}, s_{12}\}$.

As both Algorithm 5 and 6 build upon their counterparts for range query, they do not prioritize the examination of the index partitions (i.e., nodes of the interval tree or HINT partitions and their subdivisions). To this end, we devise a best-first approach, which examines the partitions in decreasing order of their potential to include the most relevant intervals to q. Algorithm 7 shows the key steps of this approach. We first find the index partitions that may include intervals which overlap with the query, but we do not access them yet. For each such partition P, we use the min/max statistics of the contained intervals to compute $UB(P)$ and $LB(P)$. We then sort the partitions in decreasing order of their $UB(P)$, breaking ties using $LB(P)$. Finally, we consider the partitions in this particular

ALGORITHM 7: *kRelQquery* best-first search

Input : Interval index $\mathcal{I}$, query interval q, number of results k
Output : k intervals that overlap with q, having the highest Rel
Output : set of all intervals that overlap with q

```
1 initialize min-priority queue Q;                    ▷ top-k list for result
2 let P contain all relevant partitions in I ;        ▷ Algorithm 1 or 2
3 sort partitions in P by UB in decreasing order; solve ties with LB;
4 foreach partition index P ∈ P do
5     if UB(P) ≤ Res(Q.top(), q) then
6         break;
7     else
8         scan P to update Q;
9 output Q;
```

Table 1: Characteristics of tested datasets

	BOOKS [8]	WEBKIT [20]	BTC [2]	TAXIS [8]
Cardinality	2,050,707	2,347,346	2,538,921	169,290,307
Size [MBs]	32	28	52	2,794
Domain	1 year	15 years	3 months	1 year
Min duration	1 hour	1 sec	1 sec	1 min
Max duration	1 year	15 years	6 days	5 hours
Avg. duration	67 days	1 year	40 mins	12 mins
Avg. duration [%]	18.4	7.22	0.03	0.002

order and access their contents (i.e., the intervals assigned to them), updating the k most relevant intervals to q so far. These intervals are again maintained inside a min-priority queue Q. As soon as the next partition's upper bound is smaller than or equal to the k-th lowest relevance to q in the current result (i.e., the relevance of $Q.top()$), we can terminate, as there is no chance for any subsequent partitions can improve the top-k results so far. Let us consider again our running example with $k = 2$ and Rel_{rq}. For the interval tree, Algorithm 7 considers the relevant nodes in the following order, v_3, v_7, v_6, v_1. After examining v_3, Q contains intervals s_{11} and s_{12}, with $Q.top() = s_{12}$. Hence, when scanning v_7 the algorithm computes $UB(v_7)$ and terminates since $UB(v_7) < Rel_{rq}(s_{12}, q)$. For HINT, Algorithm 7 orders the subdivisions of the relevant partitions as $P^{O_{in}}_{3,6}$, $P^{O_{aft}}_{3,6}$, $P^{O_{in}}_{2,3}$, $P^{O_{aft}}_{3,5}$, $P^{O_{in}}_{3,7}$, $P^{O_{in}}_{2,2}$ and $P^{R_{in}}_{2,2}$. The algorithm inserts s_{11}, s_{12} to Q after scanning the first two subdivisions $P^{O_{in}}_{3,6}$, $P^{O_{in}}_{3,6}$ and then terminates since $U(P^{O_{in}}_{2,3}) < Rel_{rq}(Q.top(), q)$.

4 EXPERIMENTAL ANALYSIS

We implemented both interval tree and HINT indices, and the query processing methods in C++, compiled using gcc (v4.8.5) with `-O3` and `-march=native` flags.[9] Our tests ran on an Apple M1 Pro (3.20GHz) with 32GBs of RAM, running MacOS 14.6.1 Sonoma.

4.1 Setup

We wrote the interval tree according to [22]. For HINT, we used its public source code[10], activating the subdivisions, the sorting and the cache misses optimizations. Similar to previous work, the datasets and the indices all reside in main memory.

[9] Code available at https://github.com/sigmod25intrel/code/
[10] https://github.com/pbour/hint/

Table 2: Overhead in space and maintenance costs for HINT

overhead	BOOKS	WEBKIT	BTC	TAXIS
space	0.02%	0.04%	2.2%	0.09%
insertions	0.3%	0.4%	3.3%	3.1%
deletions	1.2%	0.2%	5.8%	3.1%

We experimented with 4 datasets of real intervals, which have also been used in past studies; Table 1 summarizes their characteristics. BOOKS [9] contains the periods of time in 2013 when books were lent out by libraries in the city of Aarhus, Denmark.[11] WEBKIT [19] records the file history in the git repository of the Webkit project from 2001 to 2016[12]; the intervals indicate the periods during which a file did not change. BTC [2] contains historical price intervals of Bitcoin[13]; the low and high prices are used to determine the *start* and the *end* points, respectively. TAXIS [9] stores the time periods of taxi trips (pick-up and drop-off timestamps) from NY City in 2009.[14] Datasets BOOKS and WEBKIT represent inputs with long intervals, covering on average over 5% of the domain, whereas BTC and TAXIS contain short intervals, covering less than 0.1% of the domain. Lastly, the number of bits m for the HINT index on each dataset is automatically set utilizing the cost model in [14].

Our analysis focuses on the *relative* relevance definitions from Section 2.1, i.e., $Rel_r(s, q)$, $Rel_{rd}(s, q)$ and $Rel_{rq}(s, q)$. We omit $Rel_a(s, q)$ which gives the same results as $Rel_{rq}(s, q)$ if we divide its θ by $|q|$. To assess the performance of the querying methods, we measure their throughput (number of queries per second), while varying (1) the extent of the query interval as a percentage of the domain size inside $\{0.01\%, 0.0\%, 0.1\%, 0.5\%, 1\%\}$, (2) the value of θ inside $[0, 1]$ for threshold-based queries, and (3) k in $\{3, 5, 10, 50, 100\}$ for ranking queries. In each test, we run 10K random queries and vary one of the above parameters while fixing the rest to their default value (0.1% for the query extent, 0.5 for θ and 10 for k). Finally, we also assess the accuracy of the lower and upper relevance bounds in Section 3.3 from the min/max bounds in Sections 3.2.

4.2 Computing and maintaining stats & bounds

In our first set of experiments, we study the merit of the bounds employed by our framework. We start off with the min/max endpoint statistics detailed in Section 3.2 and showcase the overhead of storing and maintaining them in each partition of HINT[15] Table 2 reports the relative overhead for storing and maintaining the statistics, for each dataset. For the insertion updates, we indexed offline the first 90% of each dataset and then, added the remaining 10% of the intervals in HINT, while for the deletion updates, we removed 10% of the indexed intervals. We observe that both the space and the maintenance overheads are very low, always below 6%. The highest overhead is witnessed in BTC and TAXIS where the HINT hierarchy contains more levels (and therefore, more partitions as well) than BOOKS and WEBKIT.

We also study the accuracy of the $LB(P)$ and $UB(P)$ relevance bounds that our framework computes. Figures 7, 8 report the average absolute error for the $LB(P)$ and $UB(P)$ estimations in all four

[11] https://www.odaa.dk
[12] https://webkit.org
[13] https://www.kaggle.com/datasets/swaptr/bitcoin-historical-data
[14] https://www1.nyc.gov/site/tlc/index.page
[15] For the interval tree, there is no overhead, as the statistics can directly derived from the *Lst* and *Lend* lists in each node.

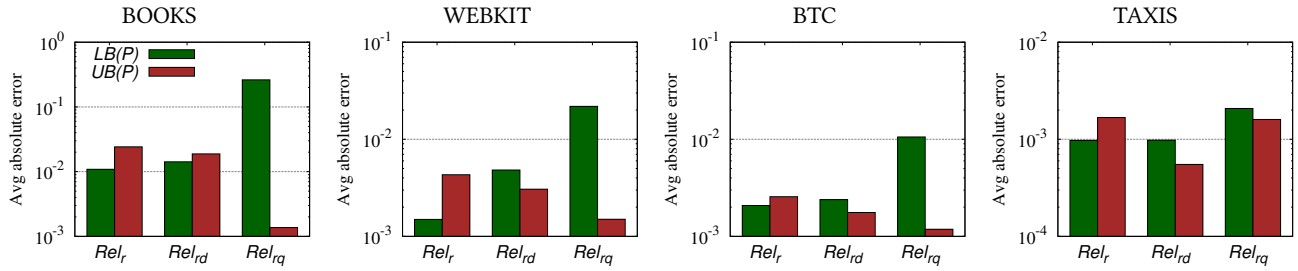

Figure 7: Accuracy of computed relevance bounds on the interval tree

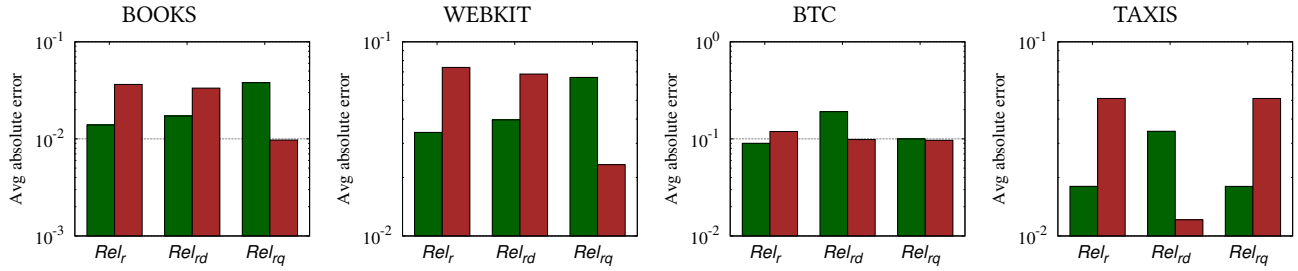

Figure 8: Accuracy of computed relevance bounds on HINT

datasets. We observe that our framework manages to estimate the highest and lowest relevance in a partition with high accuracy for both indices and under all relevance definitions; the absolute error is typically below 0.1 - recall that $Rel_r(s,q)$, $Rel_{rd}(s,q)$ and $Rel_{rq}(s,q)$ draw values inside $[0,1]$. These findings directly reflect on query performance enhancement showcased in the next experiments.

4.3 Best approach for query processing

The next set of experiments investigate the best approach, i.e., bounds and algorithm. For $\theta RelQuery$, this question translates to determining which of the lower and upper relevance bounds should be used, while for $kRelQuery$, also the best index traversal approach. We include in our analysis, a baseline which operates without any relevance bounds and simply extends the process for range selection queries (Algorithms 1 and 2); all intervals overlapping with the query are identified and then verified, as described in the beginning of Section 3. Figures 9, 10 report on the query processing performance. Overall, the tests show the benefit of using relevance bounds and the merit of our framework. The depth-first and bottom-up baselines are always outperformed by at least one method which uses bounds, in both query types, by up to 3 orders of magnitude.

For $\theta RelQuery$ (the first row of plots in each figure), we observe that utilizing $UB(P)$ typically improves more the performance over the baseline, compared to just using $LB(P)$. Exceptions arise when the queries return a large number of intervals, i.e., when Rel_{rq} is applied on datasets with longer intervals such as BOOKS and WEBKIT, where $|s| \gg |q|$ holds, or when Rel_{rq} is applied on datasets with short intervals such as BTC and TAXIS, where $|q| \gg |s|$. In these cases, the computed lower bounds approach 1, which enables the methods to massively output intervals without further comparisons and without computing their actual relevance. Nevertheless, the best option is to use both relevance bounds; such an approach successfully combines the pruning power of $UB(P)$ shown in all tests, with the advantage of $LB(P)$ in these special cases. Note that this finding applies for both the interval index and HINT.

For $kRelQuery$, the choice of bounds and method is more dataset-oriented. As a general observation, best-first processing of partitions typically benefits from using both upper and lower relevance bounds for prioritization. Yet, the best-first powered by $LB(P)$ and $UB(P)$ does not always outperform the native depth-first and bottom-up methods of interval tree and HINT, respectively, powered by $UB(P)$. Specifically, the best-first method is the fastest on datasets with long intervals; the native traversals are better on datasets with short intervals. This phenomenon is due to the number of relevant partitions per query. This number is higher in BTC, TAXIS compared to BOOKS, WEBKIT, as their indices contain more levels (compared to BOOKS) or cover a smaller domain (compared to WEBKIT). Under this, the cost of sorting the relevant partitions is slowing down best-first. In view of these findings, we use for both indices, the best-first method with $L(B)$ and $UB(P)$ on BOOKS and WEBKIT and their native method with $UB(P)$ for BTC and TAXIS.

4.4 Index comparison

Finally, we evaluate the performance of the two studied indices under their best setting. We start off by comparing them against a table-scan baseline to conceive the magnitude of the performance enhancement achieved by our framework. Table 3 shows the results for the default experimental parameters; the table clearly shows that our framework (both indices) achieves several orders of magnitude faster query processing than a straightforward approach that iterates over all input intervals and computes their relevance.

Next, we extensively compare the two indices and study how the experimental parameters of the query extent, the threshold θ and the number of request results k affect query performance. Figures 11 and 12 report the throughput for the $\theta RelQuery$ and $kRelQuery$,

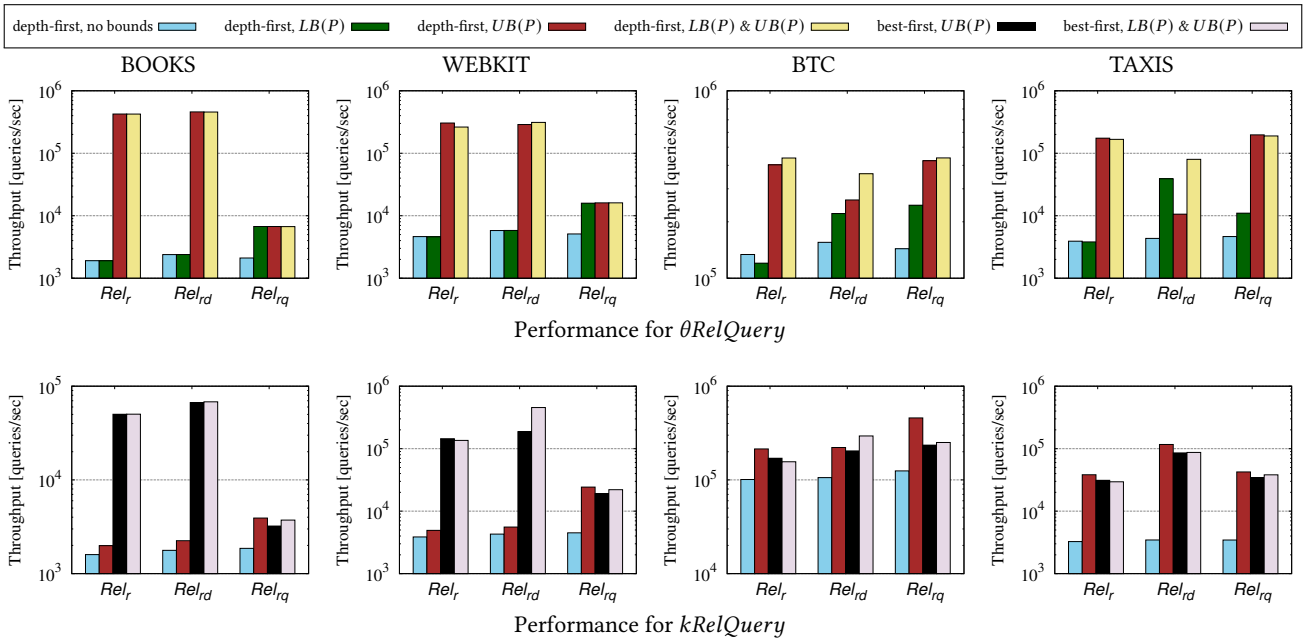

Figure 9: Query processing on the interval tree

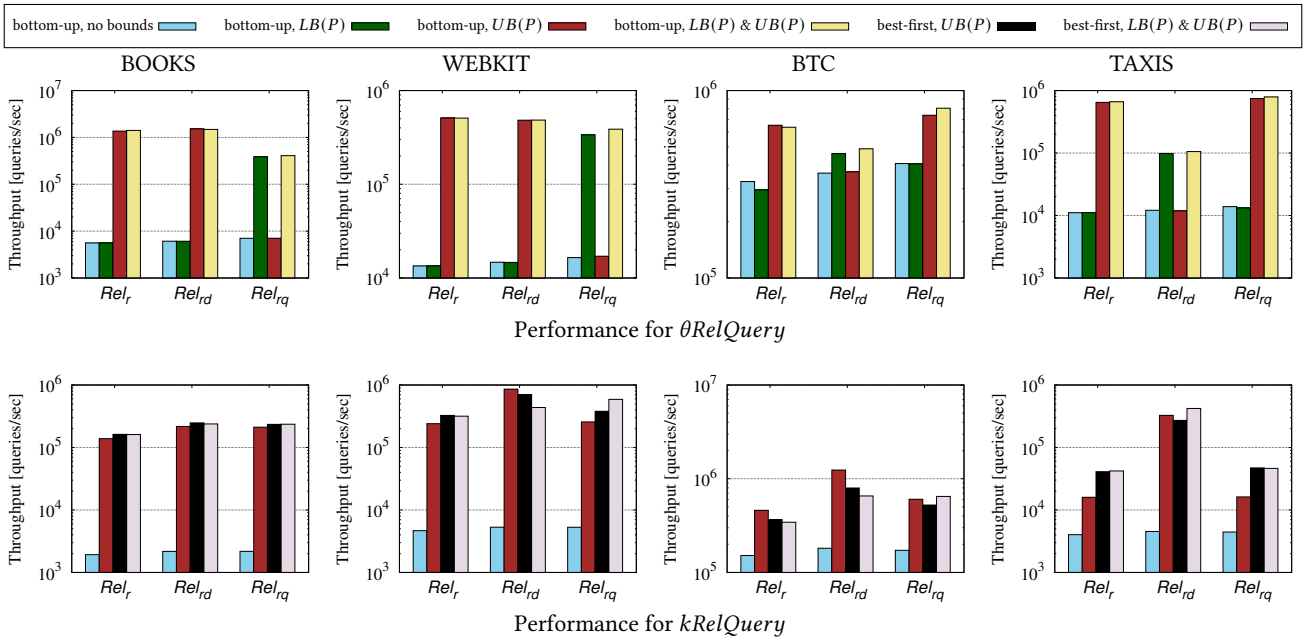

Figure 10: Query processing on HINT

respectively. First of all, we observe that query processing is negatively affected (i.e., the throughput of both methods drops) when increasing the extent of the query interval because the number of relevant partitions to be examined also rises. For $\theta RelQuery$, the methods are accelerated when the value of θ increases because in this case, the pruning power of the upper relevance bounds is enhanced and the size of result set gets smaller. In contrast, when the number of requested results increases for $kRelQuery$, the queries become more expensive and more partitions are examined to fill the result set. The extensive tests in Figures 11 and 12 also unveil that HINT is, in general, faster than the interval tree for both threshold-based and ranking relevance queries. This result aligns with the findings in [14, 16] for selection (range and stabbing) queries.

5 RELATED WORK

We discuss access methods and ranking techniques for intervals.

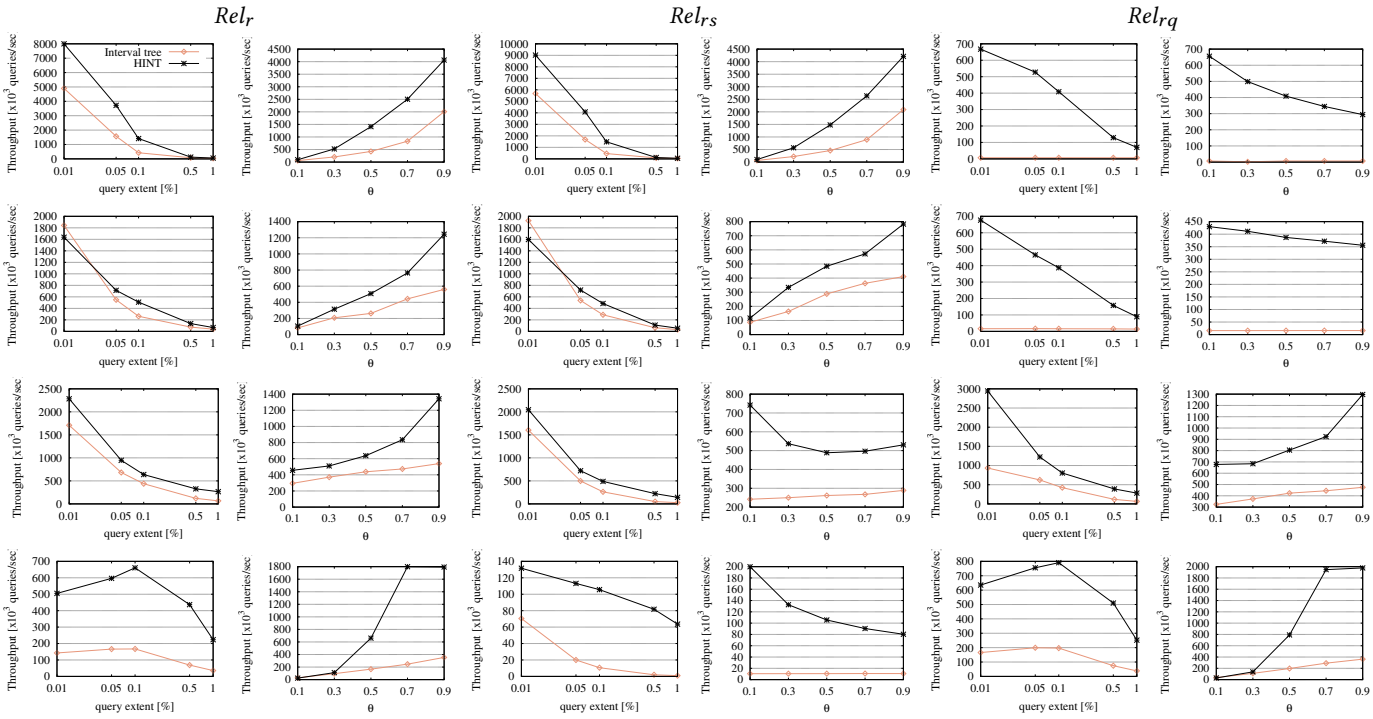

Figure 11: Index comparison: $\theta RelQuery$; first row for BOOKS, second for WEBKIT, third for BTC, fourth for TAXIS

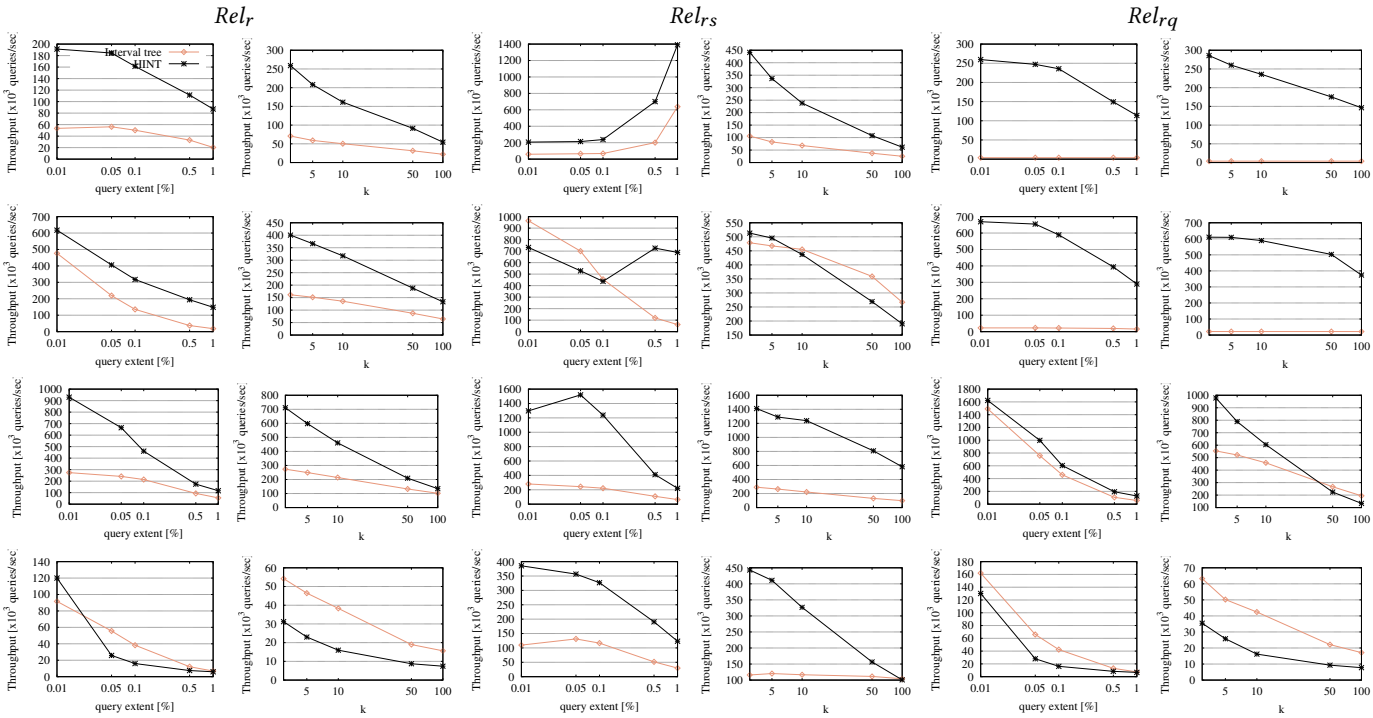

Figure 12: Index comparison: $kRelQuery$; first row for BOOKS, second for WEBKIT, third for BTC, fourth for TAXIS

Table 3: Comparison (throughput) against a table scan

BOOKS

method	$\theta RelQuery$			$kRelQuery$		
	Rel_r	Rel_{rs}	Rel_{rq}	Rel_r	Rel_{rs}	Rel_{rq}
Table scan	165			160		
Interval tree	423821	457105	6696	50280	68093	2728
HINT	1410706	1480266	408371	161300	238222	235655

WEBKIT

method	$\theta RelQuery$			$kRelQuery$		
	Rel_r	Rel_{rs}	Rel_{rq}	Rel_r	Rel_{rs}	Rel_{rq}
Table scan	584			560		
Interval tree	262768	311903	16084	135640	455143	22041
HINT	508101	484103	386997	317411	436873	588916

BTC

method	$\theta RelQuery$			$kRelQuery$		
	Rel_r	Rel_{rs}	Rel_{rq}	Rel_r	Rel_{rs}	Rel_{rq}
Table scan	917			900		
Interval tree	437744	360936	438420	214074	221829	458476
HINT	637283	488649	804783	1238550	460774	604134

TAXIS

method	$\theta RelQuery$			$kRelQuery$		
	Rel_r	Rel_{rs}	Rel_{rq}	Rel_r	Rel_{rs}	Rel_{rq}
Table scan	12			12		
Interval tree	3916	4338	4640	3258	3455	3455
HINT	11087	12125	13873	4021	4531	4436

5.1 Indexing intervals

The *interval tree* [22] (see Section 2.2.1) is a data structure that offers optimal worst-case space and time guarantees. A relational interval tree RI-tree for disk-resident data was proposed in [28]. The *segment tree* [18] is another binary search tree for intervals with $O(n \log n)$ space and time requirements, which however was designed for stabbing (or point) selection queries where the goal is to determine the intervals that contain a specific value. Simple 1D partitioning (i.e., a grid) can also be used to divide the domain into uniform partitions and replicate intervals to all partitions they overlap. To avoid duplicate results when the query range spans multiple partitions, a reference value method [21] can be used.

Other solutions for indexing intervals are the *timeline index* [27], the *period index* [6] and the *RD-index* [11]. The timeline index [27] is a general-purpose access method for temporal (versioned) data, implemented as SAP-HANA tables. A table called the event list stores a $\langle time, id, isStart \rangle$ triple for the endpoints of all intervals, where $time$ is either the start or end of an interval, specified accordingly by the boolean $isStart$ flag. In addition, at certain timestamps, called *checkpoints*, the entire set of active objects is materialized, i.e., those with an interval that contain the checkpoint. Selection queries $q = [q.start, q.end]$ are evaluated by comparing the contents of the closest checkpoint before $q.start$ and the entries in the event list after the checkpoint against the query range. The period [6] and the RD-index [11] are self-adaptive structures which split the domain into coarse partitions, and then further divide each partition hierarchically, in order to organize the contained intervals based on their positions and durations. They are specialized to range and duration queries. The RD-index [11] essentially improves upon the period index by supporting arbitrary distributions of interval durations and allowing to index the intervals either first by duration or time. Moreover, RD-index does not replicate intervals, yielding a smaller memory footprint and better query performance.

HINT was proposed in [14] and then extended in [16] to support interval range queries with arbitrary predicates from Allen's temporal algebra (e.g., *before*, *meets*, *covers*, etc.). As shown in [11, 16], HINT outperforms alternative access methods for interval range queries without other predicates (such as duration). HINT also has low space complexity in practice due to its storage optimizations.

5.2 Ranking queries over interval data

Besides range and duration selection queries, additional query types have been studied on interval data that apply some sort of ranking. Pilourdault et al. [31] define join operations that find interval pairs satisfying one of Allen's temporal relation (e.g., *meets*). Since there could be too few interval pairs satisfying this relation exactly, they accept pairs that do so approximately and assign scores to them (e.g., a pair r, s such that r ends just before s starts is better than a pair such that r ends much earlier than s). The proposed MapReduce solution for this problem cannot be used to evaluate relevance queries, as the objectives are totally different (i.e., ranking interval pairs vs. query-based retrieval, ranking based on similarity to a temporal relation vs. ranking based on the length of intersection).

Amagata [2] observed that interval overlap queries, such as "show taxis which were active between 17:00 and 22:00 yesterday", "find books which were borrowed in the last week", may retrieve too many results to be postprocessed by the user. To remedy this, Amagata [2] suggests to obtain only a small sample of the results and develop an independent range sampling technique that operates on a variant of the interval tree. In this paper, we follow a different approach, where we provide the user with an option to retrieve the *most relevant* results using overlap-based definitions of relevance.

Xu and Lu [38] consider intervals associated with weights. They study the problem of retrieving the k intervals that intersect a query interval and have the highest weights. To solve this problem they use an interval tree to retrieve the intervals that overlap with the query range and select the k best in them. Amagata et al. [3] propose a more efficient solution that extends a segment tree to sort the intervals along each path based on weight and limit the number of required accesses. Our search problem is different, because our relevance score depends on the intersection between data intervals and the query interval and not on some independent weight. Hence, the methods proposed in [3, 38] are not applicable. However, we share the same motivation that ranking of interval query results can facilitate their postprocessing and analysis.

Another related problem is the evaluation of probabilistic queries in uncertain databases [12, 37]. Assuming that values are approximated by probabilistic density functions (PDFs), given a range query, the objective is to find the values inside the query range with a probability at least θ. This is aligned with our $Rel_{rd}(s, q)$ definition, for the special case where PDFs are uniform. However, the authors in [37] and follow-up papers do not study other definitions and they focus on arbitrary multi-dimensional PDFs, proposing specialized data structures for such data, whereas our approach can be applied using off-the-shelf interval indices.

6 CONCLUSIONS

In this paper, we proposed relevance queries for intervals, which find use in many applications that manage large collections of temporal data (temporal databases, uncertain databases, etc.). Relevance queries limit the potentially numerous results of range queries that are hard to postprocess and interpret, by filtering or ranking the intervals with high relevance score to the query. We proposed a unified framework for processing queries under different definitions of relevance on any interval index that divides the intervals into partitions. At the heart of our framework lies a method for computing provable upper and lower relevance bounds for entire index partitions. We applied our framework on two popular interval indexes (interval tree and HINT); our experiments on four large real interval collections demonstrate that it achieves orders of magnitude higher throughput over a baseline approach. In the future, we plan to study the application of our framework for the efficient evaluation of top-k interval joins [31]. In addition, we plan to integrate relevance queries in PostgreSQL, which includes native ranged data types, and extend its query optimizer to consider interval-based selection and ranking predicates.

REFERENCES

[1] Mohammed Al-Kateb, Ahmad Ghazal, Alain Crolotte, Ramesh Bhashyam, Jaiprakash Chimanchode, and Sai Pavan Pakala. 2013. Temporal query processing in Teradata. In *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*. ACM, 573–578. https://doi.org/10.1145/2452376.2452443

[2] Daichi Amagata. 2024. Independent Range Sampling on Interval Data. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 449–461. https://doi.org/10.1109/ICDE60146.2024.00041

[3] Daichi Amagata, Junya Yamada, Yuchen Ji, and Takahiro Hara. 2024. Efficient Algorithms for Top-k Stabbing Queries on Weighted Interval Data. In *Database and Expert Systems Applications - 35th International Conference, DEXA 2024, Naples, Italy, August 26-28, 2024, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 14910)*. Springer, 146–152. https://doi.org/10.1007/978-3-031-68309-1_12

[4] Ahmed Awad, Riccardo Tommasini, Samuele Langhi, Mahmoud Kamel, Emanuele Della Valle, and Sherif Sakr. 2022. D^2IA: User-defined interval analytics on distributed streams. *Inf. Syst.* 104 (2022), 101679. https://doi.org/10.1016/J.IS.2020.101679

[5] 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. https://doi.org/10.1007/S007780050028

[6] 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 *Proceedings of the 16th International Symposium on Spatial and Temporal Databases, SSTD 2019, Vienna, Austria, August 19-21, 2019*. ACM, 100–109. https://doi.org/10.1145/3340964.3340965

[7] Michael H. Böhlen, Anton Dignös, Johann Gamper, and Christian S. Jensen. 2017. Temporal Data Management - An Overview. In *Business Intelligence and Big Data - 7th European Summer School, eBISS 2017, Bruxelles, Belgium, July 2-7, 2017, Tutorial Lectures (Lecture Notes in Business Information Processing, Vol. 324)*. Springer, 51–83. https://doi.org/10.1007/978-3-319-96655-7_3

[8] 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. https://doi.org/10.14778/3137628.3137644

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

[10] Ricardo Campos, Gaël Dias, Alípio Mário Jorge, and Adam Jatowt. 2014. Survey of Temporal Information Retrieval and Related Applications. *ACM Comput. Surv.* 47, 2 (2014), 15:1–15:41. https://doi.org/10.1145/2619088

[11] Matteo Ceccarello, Anton Dignös, Johann Gamper, and Christina Khnaisser. 2023. Indexing Temporal Relations for Range-Duration Queries. In *Proceedings of the 35th International Conference on Scientific and Statistical Database Management, SSDBM 2023, Los Angeles, CA, USA, July 10-12, 2023*. ACM, 3:1–3:12. https://doi.org/10.1145/3603719.3603732

[12] Reynold Cheng, Dmitri V. Kalashnikov, and Sunil Prabhakar. 2003. Evaluating Probabilistic Queries over Imprecise Data. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*. ACM, 551–562. https://doi.org/10.1145/872757.872823

[13] Reynold Cheng, Sarvjeet Singh, and Sunil Prabhakar. 2005. U-DBMS: A Database System for Managing Constantly-Evolving Data. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*. ACM, 1271–1274. http://www.vldb.org/conf/2005/papers/p1271-cheng.pdf

[14] George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2022. HINT: A Hierarchical Index for Intervals in Main Memory. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 1257–1270. https://doi.org/10.1145/3514221.3517873

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

[16] 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. https://doi.org/10.1145/3639275

[17] Nilesh N. Dalvi and Dan Suciu. 2004. Efficient Query Evaluation on Probabilistic Databases. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*. Morgan Kaufmann, 864–875. https://doi.org/10.1016/B978-012088469-8.50076-0

[18] Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. 2008. *Computational geometry: algorithms and applications, 3rd Edition*. Springer.

[19] Anton Dignös, Michael H. Böhlen, and Johann Gamper. 2014. Overlap interval partition join. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. ACM, 1459–1470. https://doi.org/10.1145/2588555.2612175

[20] 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. https://doi.org/10.1007/S00778-021-00692-3

[21] Jens-Peter Dittrich and Bernhard Seeger. 2000. Data Redundancy and Duplicate Detection in Spatial Join Processing. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*. IEEE Computer Society, 535–546. https://doi.org/10.1109/ICDE.2000.839452

[22] Herbert Edelsbrunner. 1980. *Dynamic Rectangle Intersection Searching*. Technical Report 47. Institute for Information Processing, TU Graz, Austria.

[23] 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. https://doi.org/10.1145/3588716

[24] Torsten Grust. 2002. Accelerating XPath location steps. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*. ACM, 109–120.

[25] Kun Hao, Xiaolin Zhang, Lixin Liu, and Huanxiang Zhang. 2016. An Efficient Algorithm for XML Keyword Search Based on Interval Encoding. In *13th Web Information Systems and Applications Conference, WISA 2016, Wuhan, China, September 23-25, 2016*. IEEE, 45–50. https://doi.org/10.1109/WISA.2016.19

[26] Suchen H. Hsu, Christian S. Jensen, and Richard T. Snodgrass. 1995. Valid-Time Selection and Projection. In *The TSQL2 Temporal Query Language*, Richard T. Snodgrass (Ed.). Kluwer, 249–296.

[27] 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 *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. ACM, 1173–1184. https://doi.org/10.1145/2463676.2465293

[28] Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl. 2000. Managing Intervals Efficiently in Object-Relational Databases. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*. Morgan Kaufmann, 407–418. http://www.vldb.org/conf/2000/P407.pdf

[29] Krishna G. Kulkarni and Jan-Eike Michels. 2012. Temporal features in SQL: 2011. *SIGMOD Rec.* 41, 3 (2012), 34–43. https://doi.org/10.1145/2380776.2380786

[30] Wangchao Le, Feifei Li, Yufei Tao, and Robert Christensen. 2013. Optimal splitters for temporal and multi-version databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. ACM, 109–120. https://doi.org/10.1145/2463676.2465310

[31] Julien Pilourdault, Vincent Leroy, and Sihem Amer-Yahia. 2016. Distributed Evaluation of Top-k Temporal Joins. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, 1027–1039. https://doi.org/10.1145/2882903.2882912

[32] Betty Salzberg and Vassilis J. Tsotras. 1999. Comparison of Access Methods for Time-Evolving Data. *ACM Comput. Surv.* 31, 2 (1999), 158–221. https://doi.org/10.1145/319806.319816

[33] Pierangela Samarati and Latanya Sweeney. 1998. Generalizing Data to Provide Anonymity when Disclosing Information (Abstract). In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*. ACM Press, 188. https://doi.org/10.1145/275487.275508

[34] Cynthia M. Saracco, Matthias Nicola, and Lenisha Gandhi. 2012. *A matter of time: Temporal data management in DB2 10*. Technical Report. IBM.

[35] Sarvjeet Singh, Chris Mayfield, Sagar Mittal, Sunil Prabhakar, Susanne E. Hambrusch, and Rahul Shah. 2008. Orion 2.0: native support for uncertain data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*. ACM, 1239–1242. https://doi.org/10.1145/1376616.1376744

[36] Richard T. Snodgrass and Ilsoo Ahn. 1986. Temporal Databases. *Computer* 19, 9 (1986), 35–42. https://doi.org/10.1109/MC.1986.1663327

[37] Yufei Tao, Reynold Cheng, Xiaokui Xiao, Wang Kay Ngai, Ben Kao, and Sunil Prabhakar. 2005. Indexing Multi-Dimensional Uncertain Data with Arbitrary Probability Density Functions. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*. ACM, 922–933.

[38] Jianqiu Xu and Hua Lu. 2017. Efficiently answer top-k queries on typed intervals. *Inf. Syst.* 71 (2017), 164–181. https://doi.org/10.1016/J.IS.2017.08.005

[39] Bin Zhou, Jian Pei, and Wo-Shun Luk. 2008. A brief survey on anonymization techniques for privacy preserving publishing of social network data. *SIGKDD Explor.* 10, 2 (2008), 12–22. https://doi.org/10.1145/1540276.1540279