REGULAR PAPER

HINT: a hierarchical interval index for Allen relationships

George Christodoulou[1] · Panagiotis Bouros[2] · **Nikos Mamoulis[1]**

Abstract
Indexing intervals is a fundamental problem, finding a wide range of applications, most notably in temporal and uncertain databases. We propose HINT, a novel and efficient in-memory index for range selection queries over interval collections. HINT applies a hierarchical partitioning approach, which assigns each interval to at most two partitions per level and has controlled space requirements. We reduce the information stored at each partition to the absolutely necessary by dividing the intervals in it, based on whether they begin inside or before the partition boundaries. In addition, our index includes storage optimization techniques for the effective handling of data sparsity and skewness. We show how HINT can be used to efficiently process queries based on Allen's relationships. Experiments on real and synthetic interval sets of different characteristics show that HINT is typically one order of magnitude faster than existing interval indexing methods.

Keywords Interval data · Query processing · Indexing · Main memory · Allen's algebra

1 Introduction

A wide range of applications require managing large collections of intervals. In temporal databases [6, 37], each tuple has a *validity interval*, which captures the period of time that the tuple is valid. In statistics and probabilistic databases [14], uncertain values are often approximated by (confidence or uncertainty) intervals. In data anonymization [36], attribute values are often generalized to value ranges. XML data indexing techniques [27] encode label paths as intervals and evaluate path expressions using containment relationships between the intervals. Several computational geometry problems [5] (e.g., windowing) use interval search as a module. The internal states of window queries in Stream

processors (e.g., Flink/Kafka) can be modeled and managed as intervals [2]. Event detection systems [12] represent the time periods where events are active as time intervals. Matching of event patterns as relationships between intervals is studied in [23].

We study the classic problem of indexing a large collection $\mathcal{S}$ of objects (or records), based on an interval attribute that characterizes each object. Hence, we model each object $s \in \mathcal{S}$ as a triple $\langle s.id, s.st, s.end \rangle$, where $s.id$ is the object's identifier (which can be used to access any other attribute of the object), and $[s.st, s.end]$ is the interval associated to s. Our focus is on *selection* queries, the most fundamental query type over intervals. Given a query interval $q = [q.st, q.end]$, the objective is to find the ids of all objects $s \in \mathcal{S}$, whose intervals overlap with q, i.e., they satisfy a generalized OVERLAPS (G-OVERLAPS) relationship. In addition, we study the retrieval of data intervals that satisfy one of Allen's interval algebra relationships [1] with q. Allen's algebra is used for describing precise relationships between intervals. Modeling the relative positions of temporal data finds many applications, from manufacturing processes and machine faults to business processes in general [20]. Selection queries are also known as *time travel* or *timeslice* queries in temporal databases [35]. *Stabbing* queries (*pure-timeslice* queries in temporal databases) are a special class of selection queries for which $q.st = q.end$ and the predicate is CONTAINED_BY. Without loss of gen-

✉ Panagiotis Bouros
  bouros@uni-mainz.de

George Christodoulou
gchristodoulou@cs.uoi.gr

Nikos Mamoulis
nikos@cs.uoi.gr

[1] Department of Computer Science and Engineering,
  University of Ioannina, Ioannina, Greece

[2] Institute of Computer Science, Johannes Gutenberg
  University Mainz, Mainz, Germany

erality, we assume that the intervals and queries are *closed* at both ends. [1]

Examples of selection queries include the following:

– on a relation storing employment periods: *find all employees who were employed sometime inside the* [1/1/2021, 2/28/2021] *range* (G-OVERLAPS); *find all employees who started working for a company at 1/1/2021 and stopped before 2/28/2021* (STARTS).
– on uncertain temperatures: *find all stations having temperature between 6 and 8 degrees with a nonzero probability* (G-OVERLAPS); *find all stations having temperatures, which are definitely lower/higher than 25 degrees* (BEFORE/AFTER).

For efficient selection queries over collections of intervals, classic data structures for managing intervals, like the interval tree [18], are typically used. Competitive indexing methods include the timeline index [21], 1D-grids and the period index [4]. All these methods, which we review in detail in Sect. 2, were not optimized for handling very large collections of intervals in main memory. Hence, there is room for new data structures, which exploit the characteristics and capabilities of modern machines that have large enough memory capacities for the scale of data found in most applications.

1.1 Contributions

In this paper, we propose a novel and general-purpose Hierarchical index for INTervals (HINT), suitable for applications that manage large collections of intervals. HINT defines a hierarchical decomposition of the domain and assigns each interval in $\mathcal{S}$ to at most two partitions per level. If the domain is relatively small and discrete, our index can evaluate G-OVERLAPS queries, requiring no comparisons at all. For the general case where the domain is large and/or continuous, we propose a version of HINT, denoted by $HINT^m$, which limits the number of levels to $m+1$ and greatly reduces the space requirements. $HINT^m$ conducts comparisons only for the intervals in the first and last accessed partitions at the bottom levels of the index. Some of the unique and novel characteristics of our index include:

– The intervals in each partition are further divided into groups, based on whether they begin inside or before the partition. This division (1) cancels the need for detecting and eliminating duplicate query results, (2) reduces the data accesses to the absolutely necessary, and (3) min-

imizes the space needed for storing the objects into the partitions.
– We theoretically prove that the expected number of $HINT^m$ partitions for which comparisons are necessary is at most four. This guarantees fast retrieval times, independently of the query extent and position.
– The optimized version of our index stores the intervals in all partitions at each level sequentially and uses a dedicated array with just the ids of intervals there, as well as links between non-empty partitions at each level. These optimizations facilitate sequential access to the query results at each level, while avoiding accessing unnecessary data.
– We show the necessary additional comparisons and accesses on $HINT^m$ for each relationship in Allen's algebra. In addition, we show that $HINT^m$ without the storage optimization is directly suitable for processing queries using all Allen's relationships, while maintaining the excellent performance of $HINT^m$ for G-OVERLAPS queries.
– We show how an index-based nested-loops approach for G-OVERLAPS interval joins that uses $HINT^m$ on the inner join input outperforms the state-of-the-art join method when the outer input is relatively small.

Table 1 compares HINT to previous work, based on our experiments on real and synthetic datasets. Our index is typically *one order of magnitude faster* than the competition. As we explain in Sect. 2, existing indices typically require at least one comparison for each query result (interval tree, 1D-grid) or may access and compare more data than necessary (timeline index, 1D-grid). Further, the 1D-grid, the timeline and the period index need more space than HINT in the presence of long intervals in the data due to excessive replication either in their partitions (1D-grid, period index) or their checkpoints (timeline index). HINT gracefully supports updates, as each partition (or division within a partition) is independent from others. The building cost of HINT is also low, as we verify experimentally. Overall, HINT is superior in all aspects to the state-of-the-art and constitutes an important contribution, given the fact that selection queries over large interval collections is a fundamental problem with numerous applications.

Table 1 Comparison of interval indices

Index	Query cost	Space	Updates
Interval tree [18]	Medium	Low	Slow
Timeline index [21]	Medium	Medium	Slow
1D-grid	Medium	Medium	Fast
Period index [4]	Medium	Medium	Fast
HINT/HINT^m (our work)	Low	Low	Fast

---

[1] Our index can easily be adapted to manage intervals and/or process selection queries, which are open at either or both sides, i.e., $[o.st, o.end)$, $(o.st, o.end]$ or $(o.st, o.end)$.

1.2 Comparison to our previous work

This article extends our previous work [13] in three directions. First, we elaborate on the model for tuning the value of the parameter m for HINT^m. Specifically, we include a new experiment which confirms the intuition behind our proposed model. Second, we study HINT^m performance for G-OVERLAPS interval joins. Finally, we study the evaluation of selection queries under all relationships in Allen's algebra; [13] considered only the G-OVERLAPS relationship. We show that HINT^m achieves excellent performance, independently of the query predicate.

1.3 Outline

Section 2 reviews related work and presents in detail the characteristics and weaknesses of existing interval indices. In Sect. 3, we present HINT and its generalized HINT^m version, and analyze their complexity. Focusing primarily on the G-OVERLAPS relationship, optimizations that boost the performance of HINT^m are presented in Sect. 4, and the first part of our experimental analysis on real and synthetic data against the state-of-the-art is presented in Sect. 5. Then, Sect. 6 discusses necessary changes to HINT^m for efficiently evaluating selection queries under the Allen's algebra relationships, and Sect. 7 follows up with the second part of our experiments. Finally, Sect. 8 concludes the paper with a discussion about future work.

2 Related work

In this section, we present in detail the state-of-the-art main-memory indices for intervals, to which we experimentally compare HINT in Sect. 5. In addition, we briefly discuss other relevant data structures and previous work on other queries over interval data.

2.1 Interval tree

One of the most popular data structures for intervals is Edelsbrunner's *interval tree* [18], a binary search tree, which takes $O(n)$ space and answers queries in $O(\log n + K)$ time (K is the number of query results). The tree divides the domain hierarchically by placing all intervals strictly before (after) the domain's center to the left (right) subtree and all intervals that overlap with the domain's center at the root. This process is repeated recursively for the left and right subtrees using the centers of the corresponding sub-domains. The intervals assigned to each tree node are sorted in two lists based on their starting and ending values, respectively. Interval trees are used to answer selection (i.e., *stabbing* and *range*) queries. For example, Fig. 1 shows a set of 14 inter-
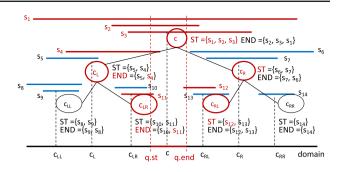

Fig. 1 Example of an interval tree

vals $s_1, \ldots, s_{14}$, which are assigned to 7 interval tree nodes and a query interval $q = [q.st, q, end]$. The domain point c corresponding to the tree's root is *contained in* the query interval; hence, all intervals in the root are reported and both the left and right children of the root have to be visited recursively. Since the left child's point c_L is before $q.st$, we access the END list from the end and report results until we find an interval s for which $s.end < q.st$; then we access recursively the right child of c_L. This process is repeated symmetrically for the root's right child c_R. The main drawback of the interval tree is that we need to perform comparisons for most of the intervals in the query result. In addition, updates on the tree can be slow because the lists at each node should be kept sorted. A relational interval tree for disk-resident data was proposed in [24].

2.2 Timeline index

The *timeline index* [21] is a general-purpose access method for temporal (versioned) data, in SAP-HANA. It keeps the endpoints of all intervals in an *event list*, which is a table of $\langle time, id, isStart \rangle$ triples, where $time$ is the value of the start or end point of the interval, id is the identifier of the interval, and $isStart$ 1 or 0, depending on whether $time$ corresponds to the start or end of the interval, respectively. The event list is sorted primarily by $time$ and secondarily by $isStart$ (descending). In addition, at certain timestamps, called *checkpoints*, the entire set of *active* object-ids is materialized, that is the intervals that contain the checkpoint. For each checkpoint, there is a link to the first triple in the event list for which $isStart$=0 and $time$ is greater than or equal to the checkpoint, Fig. 2a shows a set of five intervals $s_1, \ldots, s_5$ and Fig. 2b exemplifies a timeline index for them.

To evaluate a selection query (called *time travel* query in [21]), we first find the largest checkpoint which is smaller than or equal to $q.st$ (e.g., c_2 in Fig. 2) and initialize R as the active interval set at the checkpoint (e.g., $R = \{s_1, s_3, s_5\}$). Then, we scan the event list from the position pointed by the checkpoint, until the first triple for which $time \geq q.st$, and update R by inserting to it intervals corresponding to an $isStart = 1$ event and deleting the ones corresponding
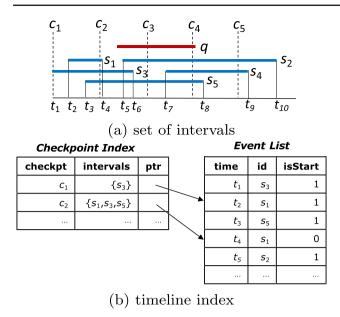
(a) set of intervals

Checkpoint Index

checkpt	intervals	ptr
c_1	$\{s_3\}$	
c_2	$\{s_1,s_3,s_5\}$	
...	...	...

Event List

time	id	isStart
t_1	s_3	1
t_2	s_1	1
t_3	s_5	1
t_4	s_1	0
t_5	s_2	1
...	...	...

(b) timeline index

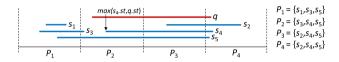**Fig. 2** Example of a timeline index

Fig. 3 Example of a 1D-grid

to a $isStart = 0$ triple (e.g., R becomes $\{s_3, s_5\}$). When we reach $q.st$, all intervals in R are guaranteed query results and they are reported. We continue scanning the event list until the first triple after $q.end$ and we add to the result the ids of all intervals corresponding to triples with $isStart = 1$ (e.g., s_2 and s_4).

The timeline index accesses more data and performs more comparisons than necessary, during query evaluation. In the worst-case scenario, where almost all intervals span almost the entire domain, all checkpoints will include almost all intervals, so the space complexity is $O(n \cdot C)$, where C is the number of checkpoints. Each query costs $O(n)$ time as $O(n)$ active intervals from a checkpoint will be read and processed. The timeline index is suitable for transaction-time temporal databases, where individual updates cost $O(1)$; however, even in this case, the amortized update cost can be as high as $O(C/n)$, if we include the construction of checkpoints.

2.3 1D-grid

A simple and practical data structure for intervals is a *1D-grid*, which divides the domain into p partitions $P_1, P_2, \ldots, P_p$. The partitions are pairwise disjoint in terms of their interval span and collectively cover the entire data domain D. Each interval is assigned to all partitions that it overlaps with. Figure 3 shows 5 intervals assigned to $p = 4$ partitions; s_1 goes

to P_1 only, while s_5 goes to all four partitions. Given a query q, the results can be obtained by accessing each partition P_i that overlaps with q. For each P_i which is *contained in q* (i.e., $q.st \leq P_i.st \wedge P_i.end \leq q.end$), all intervals in P_i are guaranteed to overlap with q. For each P_i, which overlaps with q, but is not contained in q, we should compare each $s_i \in P_i$ with q to determine whether s_i is a query result. If the interval of a query q overlaps with multiple partitions, duplicate results may be produced. An efficient approach for handling duplicates is the *reference value* method [17], which was originally proposed for rectangles but can be directly applied for 1D intervals. For each interval s found to overlap with q in a partition P_i, we compute $v = \max\{s.st, q.st\}$ as the *reference value* and report s only if $v \in [P_i.st, P_i.end]$. Since v is unique, s is reported only in one partition. In Fig. 3, interval s_4 is reported only in P_2 which contains value $\max\{s_4.st, q.st\}$.

The 1D-grid has two drawbacks. First, the duplicate results should be computed and checked before being eliminated by the reference value. Second, if the collection contains many long intervals, the index may grow large in size due to excessive replication which increases the number of duplicate results to be eliminated. In the worst-case scenario, space complexity is $O(n \cdot C)$, where C is the number of partitions and each update costs $O(C)$ time. Worst-case query cost is $O(n)$, excluding deduplication, since a query may access a partition, which includes all intervals.

2.4 Period index

The *period index* [4] is a self-adaptive structure based on domain partitioning, specialized for G-OVERLAPS and *duration* queries. The time domain is split into coarse partitions as in a 1D-grid and then each partition is divided hierarchically, in order to organize the intervals assigned to the partition based on their positions and durations. Figure 4 shows a set of intervals and how they are partitioned in a period index. There are two primary partitions P_1 and P_2, and each of them is divided hierarchically to three levels. Each level corresponds to a duration length, and each interval is assigned to the level corresponding to its duration. The top level stores intervals shorter than the length of a division there, the second level stores longer intervals but shorter than a division there, and so on. Hence, each interval is assigned to at most two divisions, except for intervals which are assigned to the bottom-most level, which can go to an arbitrary num-

Fig. 4 Example of a period index

ber of divisions. During query evaluation, only the divisions that overlap with the query interval are accessed; if the query carries a duration predicate, the divisions that are shorter than the query duration are skipped. For `G-OVERLAPS` queries, the period index performs in par with the interval tree and the 1D-grid [4], so we also compare against this index in Sect. 5. In the worst case, space complexity is $O(n \cdot C)$, where C is the number of coarse partitions and each query and update costs $O(C)$ time (i.e., same as the 1D-grid).

2.5 Other indexing works

Another classic data structure for intervals is the *segment tree* [5], a binary search tree with $O(n \log n)$ space complexity that answers stabbing queries in $O(\log n + K)$ time. The segment tree is not designed for `G-OVERLAPS` queries, for which it requires a duplicate result elimination mechanism. In computational geometry [5], indexing intervals was studied as a subproblem within orthogonal 2D range search; typically, the worst-case optimal interval tree is used. Indexing intervals re-gained interest with the advent of temporal databases [6]. For temporal data, a number of indices are proposed for secondary memory, mainly for effective versioning and compression [3, 26]. Such indices are tailored for historical versioned data, while we focus on arbitrary interval sets, queries, and updates.

2.6 Interval joins and aggregation

Additional research on indexing intervals does not address selection queries, but other operations such as *temporal aggregation* [21, 22, 29] and *interval joins* [7, 8, 10–12, 15, 16, 33, 34, 38]. The timeline index [21] can be directly used for temporal aggregation. Piatov et al. [32] presented plane-sweep algorithms that extend the timeline index to support aggregation over fixed intervals, sliding window aggregates, and MIN/ MAX aggregates. Timeline was later adapted for interval overlap joins [33, 34]. In Sect. 5.4.1, we consider our proposed indexing for join computation in an index-based nested-loops fashion, and compare it against the state-of-the-art algorithm optFS from [10]. Similar to previous work, optFS builds on a highly optimized variant of plane-sweep to join un-indexed collections of intervals. A *domain partitioning* technique for parallel processing of interval joins was proposed in [7, 8, 10]. Alternative partitioning techniques for interval joins were proposed in [11, 15]. Partitioning techniques for interval joins cannot replace interval indices as they are not designed for selection queries. Temporal joins on Allen's algebra relationships for RDF data were studied in [12]. Multi-way interval joins in the context of temporal k-clique enumeration were studied in [38]. Awad et al. [2] define *interval events* of the same or different types that are observed in succession in data streams. Analytical operations based on aggregation or reasoning can be used to formulate composite interval events.

Table 2 Table of notation

Notation	Description
$s.id, s.st, s.end$	Identifier, start, end point of interval s
$q = [q.st, q.end]$	Query interval
$prefix(k, x)$	k-Bit prefix of integer x
$P_{\ell,i}$	i-th Partition at level ℓ of HINT/HINT^m
$P_{\ell,f}$ ($P_{\ell,l}$)	First (last) partition at level ℓ overlapping with q
$P_{\ell,i}^O$ ($P_{\ell,i}^R$)	Sub-partition of $P_{\ell,i}$ with originals (replicas)
$P_{\ell,i}^{O_{in}}$ ($P_{\ell,i}^{O_{aft}}$)	Intervals in $P_{\ell,i}^O$ ending inside (after) $P_{\ell,i}$
$P_{\ell,i}^{R_{in}}$ ($P_{\ell,i}^{R_{aft}}$)	Intervals in $P_{\ell,i}^R$ ending inside (after) $P_{\ell,i}$

3 HINT

In this section, we propose the *Hierarchical index for INTervals* or HINT, which defines a hierarchical domain decomposition and assigns each interval to at most two partitions per level. The primary goal of the index is to minimize the number of comparisons during query evaluation, while keeping the space requirements relatively low, even when there are long intervals in the collection. HINT applies a smart division of intervals in each partition into two groups, which avoids the production and handling of duplicate query results and minimizes the number of accessed intervals. In Sect. 3.1, we present a version of HINT, which avoids comparisons overall during query evaluation, but it is not always applicable and may have high space requirements. Section 3.2 presents HINT^m, the general version of our index, used for intervals in arbitrary domains. Last, Sect. 3.3 describes our analytical model for setting the m parameter and Sect. 3.4 discusses updates. Table 2 summarizes the notation used in the paper.

3.1 A comparison-free version of HINT

We first describe a version of HINT, which is appropriate for a *discrete* and *not very large* domain. Specifically, assume that the domain D wherefrom the endpoints of intervals in S take value is $[0, 2^m - 1]$. We can define a regular hierarchical decomposition of D into partitions, where at each level ℓ from 0 to m, there are 2^ℓ partitions, denoted by array $P_{\ell,0}, \ldots, P_{\ell,2^\ell-1}$. Figure 5 illustrates the hierarchical domain partitioning for $m = 4$.

Each interval $s \in S$ is assigned to the *smallest set of partitions* from all levels which collectively define s. It is not hard to show that s will be assigned to at most two partitions per level. For example, in Fig. 5, interval $[5, 9]$ is assigned to one partition at level $\ell = 4$ and two partitions at level $\ell = 3$.

ALGORITHM 1: Assignment of an interval to partitions

Input : HINT index $\mathcal{H}$, interval s
Output : updated $\mathcal{H}$ after indexing s

1 $a \leftarrow s.st; b \leftarrow s.end;$ ▷ set masks to s endpoints
2 $\ell \leftarrow m;$ ▷ start at the bottom-most level
3 **while** $\ell \geq 0$ **and** $a \leq b$ **do**
4   **if** *last bit of a is 1* **then**
5   **add** s to $\mathcal{H}.P_{\ell,a};$ ▷ update partition
6      $a \leftarrow a + 1;$
7   **if** *last bit of b is 0* **then**
8   **add** s to $\mathcal{H}.P_{\ell,b};$ ▷ update partition
9      $b \leftarrow b - 1;$
10    $a \leftarrow a \div 2; b \leftarrow b \div 2;$ ▷ cut-off last bit
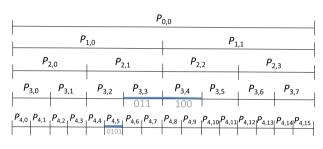11 $\ell \leftarrow \ell - 1;$ ▷ repeat for previous level

---

ALGORITHM 2: Searching HINT

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

1 $\mathcal{R} \leftarrow \emptyset;$
2 **foreach** *level ℓ in $\mathcal{H}$* **do**
3    $p \leftarrow prefix(\ell, q.st);$ ▷ 1st relevant partition
4    $\mathcal{R} \leftarrow \mathcal{R} \cup \{s.id | s \in \mathcal{H}.P^O_{\ell,p} \cup \mathcal{H}.P^R_{\ell,p}\}$
5   **while** $p < prefix(\ell, q.end)$ **do**
6      $p \leftarrow p + 1;$ ▷ next relevant partition
7      $\mathcal{R} \leftarrow \mathcal{R} \cup \{s.id | s \in \mathcal{H}.P^O_{\ell,p}\}$
8 **return** $\mathcal{R};$

Fig. 5 Hierarchical partitioning and assignment of [5,9]

Fig. 6 Accessed partitions for query [5,9]

The assignment procedure is described by Algorithm 1. In a nutshell, for an interval $[a, b]$, starting from the bottom-most level ℓ, if the last bit of a (resp. b) is 1 (resp. 0), we assign the interval to partition $P_{\ell,a}$ (resp. $P_{\ell,b}$) and increase a (resp. decrease b) by one. We then update a, b by cutting-off their last bits (i.e., integer division by 2, or bitwise right-shift). If, at the next level, $a > b$ holds, indexing $[a, b]$ is done.

3.1.1 Query evaluation

A selection query q can be evaluated by finding at each level the partitions that overlap with q. Specifically, the partitions that overlap with the query interval q at level ℓ are partitions $P_{\ell, prefix(\ell, q.st)}$ to $P_{\ell, prefix(\ell, q.end)}$, where $prefix(k, x)$ denotes the k-bit prefix of integer x. We call these partitions *relevant* to the query q. All intervals in the relevant partitions are guaranteed to overlap with q and intervals in none of these partitions cannot possibly overlap with q. However, since the same interval s may exist in multiple partitions that overlap with a query, s may be reported multiple times in the query result.

We propose a technique that avoids the production and therefore, the need for elimination of duplicates and, at the same time, minimizes the number of data accesses. For this, we divide the intervals in each partition $P_{\ell,i}$ into two groups: *originals* $P^O_{\ell,i}$ and *replicas* $P^R_{\ell,i}$. Group $P^O_{\ell,i}$ contains all intervals $s \in P_{\ell,i}$ that *begin* at $P_{\ell,i}$, i.e., $prefix(\ell, s.st) = i$. Group $P^R_{\ell,i}$ contains all intervals $s \in P_{\ell,i}$ that begin before

$P_{\ell,i}$, i.e., $prefix(\ell, s.st) \neq i$. [2] Each interval is added as original in only one partition of HINT. For example, interval [5, 9] in Fig. 5 is added to $P^O_{4,5}$, $P^R_{3,3}$, and $P^R_{3,4}$.

Given a query q, at each level ℓ of the index, we report all intervals in the first relevant partition $P_{\ell,f}$ (i.e., $P^O_{\ell,f} \cup P^R_{\ell,f}$). Then, for every other relevant partition $P_{\ell,i}, i > f$, we report all intervals in $P^O_{\ell,i}$ and ignore $P^R_{\ell,i}$. This guarantees that no result is missed and no duplicates are produced. The reason is that each interval s will appear as original in just one partition; hence, reporting only originals cannot produce any duplicates. At the same time, all replicas $P^R_{\ell,f}$ in the first partitions per level ℓ that overlap with q begin *before* q and overlap with q, so they should be reported. On the other hand, replicas $P^R_{\ell,i}$ in subsequent relevant partitions ($i > f$) contain intervals, which are either originals in a previous partition $P^R_{\ell,j}, j < i$ or replicas in $P^R_{\ell,f}$, so, they can safely be skipped. Algorithm 2 describes the search algorithm using HINT.

For example, consider the hierarchical partitioning of Fig. 6 and a query interval $q = [5, 9]$. The binary representations of $q.st$ and $q.end$ are 0101 and 1001, respectively. The relevant partitions at each level are shown in bold (blue) and dashed (red) lines and can be determined by the corresponding prefixes of 0101 and 1001. At each level ℓ, *all*

---

[2] Whether an interval $s \in P_{\ell,i}$ is assigned to $P^O_{\ell,i}$ or $P^R_{\ell,i}$ is determined at insertion time (Algorithm 1). At the first time Line 5 is executed, s is added as an original and in all other cases as a replica. If Line 5 is never executed, then s is added as original the only time that Line 8 is executed.

intervals (both originals and replicas) in the first partitions $P_{\ell,f}$ (bold/blue) are reported while in the subsequent partitions (dashed/red), *only* the *original* intervals are.

Discussion The version of HINT described above finds all query results, without any comparisons. Hence, in each partition $P_{\ell,i}$, we only have to keep the ids of the intervals that are assigned to $P_{\ell,i}$ and do not have to store/replicate the interval endpoints. Further, the relevant partitions at each level are computed by fast bit-shifting operations which are comparison-free. Under this, we expect a pipelined execution as CPU branch mispredictions are reduced. To use HINT for arbitrary integer domains, we should first normalize all interval endpoints by subtracting the minimum endpoint, in order to convert them to values in a $[0, 2^m - 1]$ domain (the same transformation should be applied on the queries). If the required m is very large, we can index the intervals based on their m-bit prefixes and support approximate search on discretized data. Approximate search can also be applied on intervals in a real-valued domain, after rescaling and discretization in a similar way.

3.2 HINT^m: indexing arbitrary intervals

We now present a generalized version of HINT, denoted by HINT^m, which can be used for intervals in arbitrary domains. HINT^m uses a hierarchical domain partitioning with $m + 1$ levels, based on a $[0, 2^m - 1]$ domain D; each raw interval endpoint is *mapped* to a value in D, by linear rescaling. The mapping function $f(\mathbb{R} \rightarrow D)$ is $f(x) = \lfloor \frac{x - min(x)}{max(x) - min(x)} \cdot (2^m - 1) \rfloor$, where $min(x)$ and $max(x)$ are the minimum and maximum interval endpoints in the dataset S, respectively. Each raw interval $[s.st, s.end]$ is mapped to interval $[f(s.st), f(s.end)]$. The mapped interval is then assigned to at most two partitions per level in HINT^m, using Algorithm 1.

For the ease of presentation, we will assume that the raw interval endpoints take values in $[0, 2^{m'} - 1]$, where $m' > m$, which means that the mapping function f simply outputs the m most significant bits of its input. As an example, assume that $m = 4$ and $m' = 6$. Interval [21, 38] (=[0b010101, 0b100110]) is mapped to interval [5, 9] (=[0b0101, 0b1001]) and assigned to partitions $P_{4,5}$, $P_{3,3}$, and $P_{3,4}$, as shown in Fig. 5.

So, in contrast to HINT, the set of partitions whereto an interval s is assigned in HINT^m does not define s, but the smallest interval in the $[0, 2^m - 1]$ domain D, which *covers* s. As in HINT, at each level ℓ, we divide each partition $P_{\ell,i}$ to $P_{\ell,i}^O$ and $P_{\ell,i}^R$, to avoid duplicate results.

3.2.1 Query evaluation using HINT^m

For a query q, simply reporting all intervals in the relevant partitions at each level (as in Algorithm 2) would produce
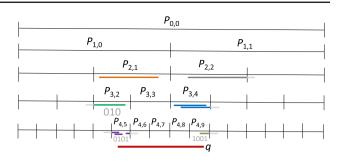

Fig. 7 Avoiding redundant comparisons in HINT^m

false positives. Instead, comparisons to the query endpoints may be required for the first and the last partition at each level that overlap with q. Specifically, we can consider each level of HINT^m as a 1D-grid (see Sect. 2) and go through the partitions at each level ℓ that overlap with q. For the first partition $P_{\ell,f}$, we verify whether s overlaps with q for each interval $s \in P_{\ell,f}^O$ and each $s \in P_{\ell,f}^R$. For the last partition $P_{\ell,l}$, we verify whether s overlaps with q for each interval $s \in P_{\ell,l}^O$. For each partition $P_{\ell,i}$ between $P_{\ell,f}$ and $P_{\ell,l}$, we report all $s \in P_{\ell,i}^O$ without any comparisons. As an example, consider the HINT^m index and the query interval q shown in Fig. 7. The identifiers of the relevant partitions to q are shown in the figure (and also some indicative intervals that are assigned to these partitions). At level $m = 4$, we have to perform comparisons for all intervals in the first relevant partitions $P_{4,5}$. In partitions $P_{4,6},...,P_{4,8}$, we just report the originals in them as results, while in partition $P_{4,9}$ we compare the start points of all originals with q, before we can confirm whether they are results or not. We can simplify the overlap tests at the first and the last partition of each level ℓ based on the following:

Lemma 1 *At every level ℓ, each $s \in P_{\ell,f}^R$ is a query result iff $q.st \leq s.end$. If $l > f$, each $s \in P_{\ell,l}^O$ is a query result iff $s.st \leq q.end$.*

Proof For the first relevant partition $P_{\ell,f}$ at each level ℓ, for each replica $s \in P_{\ell,f}^R$, $s.st < q.st$, so $q.st \leq s.end$ suffices as an overlap test. For the last partition $P_{\ell,l}$, if $l > f$, for each original $s \in P_{\ell,f}^O$, $q.st < s.st$, so $s.st \leq q.end$ suffices as an overlap test. □

3.2.2 Avoiding redundant comparisons

One of our most important findings in this study and a powerful feature of HINT^m is that at most levels, it is not necessary to do comparisons at the first and/or the last partition. For instance, in the previous example, we do not have to perform comparisons for partition $P_{3,4}$, since any interval assigned to $P_{3,4}$ should overlap with $P_{4,8}$ and the interval spanned by $P_{4,8}$ is covered by q. This means that the start point of all

intervals in $P_{3,4}$ is guaranteed to be before $q.end$ (which is inside $P_{4,9}$). In addition, observe that for any relevant partition which is the last partition at an upper level and covers $P_{3,4}$ (i.e., partitions $\{P_{2,2}, P_{1,1}, P_{0,0}\}$), we do not have to conduct the $s.st \leq q.end$ tests as intervals in these partitions are guaranteed to start before $P_{4,9}$. The lemma below formalizes these observations:

Lemma 2 *If the first (resp. last) relevant partition for a query q at level ℓ ($\ell < m$) starts (resp. ends) at the same value as the first (resp. last) relevant partition at level $\ell + 1$, then for every first (resp. last) relevant partition $P_{v,f}$ (resp. $P_{v,l}$) at levels $v \leq \ell$, each interval $s \in P_{v,f}$ (resp. $s \in P_{v,l}$) satisfies $s.end \geq q.st$ (resp. $s.st \leq q.end$).*

Proof Let $P.st$ (resp. $P.end$) denote the first (resp. last) domain value of partition P. Consider the first relevant partition $P_{\ell,f}$ at level ℓ and assume that $P_{\ell,f}.st = P_{\ell+1,f}.st$. Then, for every interval $s \in P_{\ell,f}$, $s.end \geq P_{\ell+1,f}.end$, otherwise s would have been allocated to $P_{\ell+1,f}$ instead of $P_{\ell,f}$. Further, $P_{\ell+1,f}.end \geq q.st$, since $P_{\ell+1,f}$ is the first partition at level $\ell + 1$ which overlaps with q. Hence, $s.end \geq q.st$. Moreover, for every interval $s \in P_{v,f}$ with $v < \ell$, $s.end \geq P_{\ell+1,f}.end$ holds, as interval $P_{v,f}$ covers interval $P_{\ell,f}$; so, we also have $s.end \geq q.st$. Symmetrically, we prove that if $P_{\ell,l}.end = P_{\ell+1,l}.end$, then for each $s \in P_{v,l}$, $v \leq \ell$, $s.st \leq q.end$. □

We next focus on how to rapidly check the condition of Lemma 2. Essentially, if the last bit of the offset f (resp. l) of the first (resp. last) partition $P_{\ell,f}$ (resp. $P_{\ell,l}$) relevant to the query at level ℓ is 0 (resp. 1), then the first (resp. last) partition at level $\ell - 1$ above satisfies the condition. For example, in Fig. 7, consider the last relevant partition $P_{4,9}$ at level 4. The last bit of $l = 9$ is 1; so, the last partition $P_{3,4}$ at level 3 satisfies the condition and we do not have to perform comparisons in the last partitions at level 3 and above.

Algorithm 3 is a pseudocode for HINT^m search. The algorithm accesses all index levels, bottom-up. It uses two auxiliary flags *compfirst* and *complast* to mark whether it is necessary to perform comparisons at the current level (and all levels above it) at the first and the last partition, respectively, according to the discussion in the previous paragraph. At each level ℓ, we find the offsets of the relevant partitions to the query, based on the ℓ-prefixes of $q.st$ and $q.end$ (Line 4). For the first position $f = prefix(q, st)$, the partitions holding originals and replicas $P_{\ell,f}^O$ and $P_{\ell,f}^R$ are accessed. The algorithm first checks whether $f = l$, i.e., the first and the last partitions coincide. In this case, if *compfirst* and *complast* are set, then we perform all comparisons in $P_{\ell,f}^O$ and apply the first observation in Lemma 1 to $P_{\ell,f}^R$. Else, if only *complast* is set, we can safely skip the $q.st \leq s.end$ comparisons; if only *compfist* is set, regardless whether $f = l$, we just perform $q.st \leq s.end$ comparisons to both originals and replicas

---

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

1   $compfirst \leftarrow TRUE$; $complast \leftarrow TRUE$;
2   $\mathcal{R} \leftarrow \emptyset$;
3   **for** $\ell = m$ **to** 0 **do**   ▷ bottom-up
4     $f \leftarrow prefix(\ell, q.st)$; $l \leftarrow prefix(\ell, q.end)$;
5   **for** $i = f$ **to** l **do**
6   **if** $i = f$ **then**   ▷ 1st relevant partition
7   **if** $i = l$ **and** $compfirst$ **and** $complast$ **then**
8           $\mathcal{R} \leftarrow \mathcal{R} \cup \{s.id | s \in \mathcal{H}.P_{\ell,i}^O, q.st \leq s.end \wedge s.st \leq q.end\}$;
9           $\mathcal{R} \leftarrow \mathcal{R} \cup \{s.id | s \in \mathcal{H}.P_{\ell,i}^R, q.st \leq s.end\}$;
10   **else if** $i = l$ **and** $complast$ **then**
11          $\mathcal{R} \leftarrow \mathcal{R} \cup \{s.id | s \in \mathcal{H}.P_{\ell,i}^O, s.st \leq q.end\}$;
12          $\mathcal{R} \leftarrow \mathcal{R} \cup \{s.id | s \in \mathcal{H}.P^R\}$;
13   **else if** $compfirst$ **then**
14          $\mathcal{R} \leftarrow \mathcal{R} \cup \{s.id | s \in \mathcal{H}.P_{\ell,i}^O \cup \mathcal{H}.P_{\ell,i}^R, q.st \leq s.end\}$;
15   **else**
16          $\mathcal{R} \leftarrow$ '$\mathcal{R} \cup \{s.id | s \in \mathcal{H}.P_{\ell,i}^O \cup \mathcal{H}.P_{\ell,i}^R\}$;
17   **else if** $i = l$ **and** $complast$ **then**   ▷ last partition, $l > f$
18        $\mathcal{R} \leftarrow \mathcal{R} \cup \{s.id | s \in \mathcal{H}.P_{\ell,i}^O, s.st \leq q.end\}$;
19   **else**   ▷ in-between or last ($l > f$), no comparisons
20        $\mathcal{R} \leftarrow \mathcal{R} \cup \{s.id | s \in \mathcal{H}.P_{\ell,i}^O\}$;
21   **if** $f \bmod 2 = 0$ **then**   ▷ last bit of f is 0
22       $compfirst \leftarrow FALSE$;
23   **if** $l \bmod 2 = 1$ **then**   ▷ last bit of l is 1
24       $complast \leftarrow FALSE$;
25 **return** $\mathcal{R}$;

---

in the first partition. If neither *compfirst* nor *complast* are set, we just report all intervals in the first partition as results. If we are at the last partition $P_{\ell,l}$ and $l > f$ (Line 17) then we just examine $P_{\ell,l}^O$ and apply just the $s.st \leq q.end$ test for each interval there, according to Lemma 1. Last, for all partitions in-between the first and the last, we simply report all original intervals there.

3.2.3 Complexity analysis

Let n be the number of intervals in $\mathcal{S}$. Assume that the domain is $[0, 2^{m'} - 1]$, with $m' > m$. To analyze the space complexity of HINT^m, we first prove that:

Lemma 3 *The total number of intervals assigned at the lowest level m of HINT^m is expected to be n.*

Proof Each interval $s \in \mathcal{S}$ will go to zero, one, or two partitions at level m, based on the bits of $s.st$ and $s.end$ at position m (see Algorithm 1); on average, s will go to one partition. □

Using Algorithm 1, when an interval is assigned to a partition at a level ℓ, the interval is *truncated* (i.e., shortened) by $2^{m'-\ell}$. Based on this, we analyze the space complexity of HINT^m as follows.

Theorem 1 *Let λ be the average length of intervals in input collection S. The space complexity of HINT^m is $O(n \cdot \log(2^{\log \lambda - m' + m} + 1))$.*

Proof Based on Lemma 3, each $s \in S$ will be assigned on average to one partition at level m and will be truncated by $2^{m'-m}$. Following Algorithm 1, at the next level $m - 1$, s is also be expected to be assigned to one partition (see Lemma 3) and truncated by $2^{m'-m+1}$, and so on, until the entire interval is truncated (condition $a \leq b$ is violated at Line 3 of Algorithm 1). Hence, we are looking for the number of levels whereto each s will be assigned, or for the smallest k for which $2^{m'-m} + 2^{m'-m+1} + \cdots + 2^{m'-m+k-1} \geq \lambda$. Solving the inequality gives $k \geq \log(2^{\log \lambda - m' + m} + 1)$ and the space complexity of HINT^m is $O(n \cdot k)$. □

For the computational cost of queries in terms of conducted comparisons, in the worst case, $O(n)$ intervals are assigned to the first relevant partition $P_{m,f}$ at level m and $O(n)$ comparisons are required. To estimate the *expected cost* of query evaluation in terms of conducted comparisons, we assume a uniform distribution of intervals to partitions and random query intervals.

Lemma 4 *The expected number of HINT^m partitions for which we have to conduct comparisons is four.*

Proof At the last level of the index m, we definitely have to do comparisons in the first and the last partition (which are different in the worst case). At level $m - 1$, for each of the first and last partitions, we have a 50% chance to avoid comparisons, due to Lemma 2. Hence, the expected number of partitions for which we have to perform comparisons at level $m - 1$ is 1. Similarly, at level $m - 2$ each of the yet active first/last partitions has a 50% chance to avoid comparisons. Overall, for the worst-case conditions, where m is large and q is long, the expected number of partitions, for which we need to perform comparisons is $2 + 1 + 0.5 + 0.25 + \cdots = 4$. □

Theorem 2 *The expected number of comparisons during query evaluation over HINT^m is $O(n/2^m)$.*

Proof For each query, we conduct comparisons at least in the first and the last relevant partitions at level m. The expected number of intervals, in each of these two partitions, is $O(n/2^m)$, considering Lemma 3 and assuming a uniform distribution of the intervals in the partitions. In addition, due to Lemma 4, the number of expected additional partitions that require comparisons is 2 and each of these two partitions is

expected to also hold at most $O(n/2^m)$ intervals, by Lemma 3 on the levels above m and using the truncated intervals after their assignment to level m (see Algorithm 1). Hence, q is expected to be compared with $O(n/2^m)$ intervals in total and the cost of each such comparison is $O(1)$. □

In the worst case, all data intervals fall at the top-most level $\ell = 0$ and the queries fall inside $[2^m - 2, 2^m - 1]$; in this (extreme) case, query cost is $O(n)$, as all intervals are compared with each query.

3.3 Setting *m*

As shown in Sect. 3.2.3, the space requirements and the search performance of HINT^m depend on the value of m. For large values of m, the cost of accessing comparison-free results will dominate the computational cost of comparisons. We conduct an analytical study for estimating m_{opt}: the smallest value of m, which is expected to result in a HINT^m of search performance close to the best possible, while achieving the lowest possible space requirements. Our study uses simple statistics namely, the number of intervals $n = |\mathcal{S}|$, the mean length λ_s of data intervals and the mean length λ_q of query intervals. We assume that the endpoints and the lengths of intervals and queries are uniformly distributed.

The overall cost of query evaluation consists of (1) the cost for determining the relevant partitions per level, denoted by C_p, (2) the cost of conducting comparisons between data intervals and the query, denoted by C_{cmp}, and (3) the cost of accessing query results in the partitions for which we do not have to conduct comparisons, denoted by C_{acc}. Cost C_p is negligible, as the partitions are determined by a small number m of bit-shifting operations. To estimate C_{cmp}, we need to estimate the number of intervals in the partitions whereat we need to conduct comparisons and multiply this by the expected cost β_{cmp} per comparison. To estimate C_{acc}, we need to estimate the number of intervals in the corresponding partitions and multiply this by the expected cost β_{acc} of (sequentially) accessing and reporting one interval. β_{cmp} and β_{acc} are machine-dependent and can easily be estimated by experimentation.

According to Algorithm 3, unless λ_q is smaller than the length of a partition at level m, there will be two partitions that require comparisons at level m, one partition at level $m - 1$, etc. with the expected number of partitions being at most four (see Lemma 4). Hence, we can assume that C_{cmp} is practically dominated by the cost of processing two partitions at the lowest level m. As each partition at level m is expected to have $n/2^m$ intervals (see Lemma 3), we have $C_{cmp} = \beta_{cmp} \cdot n/2^m$. Then, the number of accessed intervals for which we expect to apply no comparisons is $|Q| - 2 \cdot n/2^m$, where $|Q|$ is the total number of expected query results. Under this, we have $C_{acc} = \beta_{acc} \cdot (|Q| - 2 \cdot n/2^m)$. We can estimate $|Q|$ using the selectivity analysis for

(multidimensional) intervals and queries in [31] as $|Q| = n \cdot \frac{\lambda_s + \lambda_q}{\Lambda}$, where Λ is the length of the entire domain with all intervals in $\mathcal{S}$ (i.e., $\Lambda = \max_{\forall s \in \mathcal{S}} s.end - \min_{\forall s \in \mathcal{S}} s.st$).

With C_{cmp} and C_{acc}, we now estimate m_{opt}. First, we gradually increase m from 1 to its max value m' (determined by Λ), and compute the expected cost $C_{cmp} + C_{acc}$. For $m = m'$, HINT^m corresponds to the comparison-free HINT with the lowest expected cost. Then, we select as m_{opt} the lowest value of m for which $C_{cmp} + C_{acc}$ converges to the cost of the $m = m'$ case.

3.4 Updates

We handle insertions to an existing HINT/HINT^m by calling Algorithm 1 for each new interval s. Small adjustments are needed for HINT^m to add s to the originals division at the first partition assignment, i.e., to $P_{\ell,a}^O$ or $P_{\ell,b}^O$, and to the replicas division for every other partition, i.e., to $P_{\ell,a}^R$ or $P_{\ell,b}^R$. Further, we handle deletions using tombstones, similarly to previous studies [25, 30] and recent indexing approaches [19]. Given an interval s for deletion, we first search the index to locate all partitions that contain s (both as original and as replica) and then, replace $s.id$ by a special "tombstone" id to signal the logical deletion. Each insertion costs $O(m)$ time as an interval is added to up to $2m$ partitions, and finding the partitions at each level costs $O(1)$ time. By running the same algorithm, we find the partitions that include an interval to be deleted in $O(m)$ time. Last, we handle modifications to an existing interval, via a deletion and a consecutive insertion.

4 Optimizing HINT^m

In this section, we discuss optimization techniques, which greatly improve the performance of HINT^m (and HINT) in practice. First, we show how to reduce the number of partitions in HINT^m where comparisons are performed and how to avoid accessing unnecessary data. Next, we show how to handle very sparse or skewed data at each level of HINT/HINT^m. Another optimization is decoupling the storage of the interval ids with the storage of interval endpoints in each partition. Finally, we revisit updates under the prism of these optimizations.

4.1 Subdivisions and space decomposition

Recall that, at each level ℓ of HINT^m, every partition $P_{\ell,i}$ is divided into $P_{\ell,i}^O$ (holding originals) and $P_{\ell,i}^R$ (holding replicas). We propose to further divide each $P_{\ell,i}^O$ into $P_{\ell,i}^{O_{in}}$ and $P_{\ell,i}^{O_{aft}}$, so that $P_{\ell,i}^{O_{in}}$ (resp. $P_{\ell,i}^{O_{aft}}$) holds the intervals from
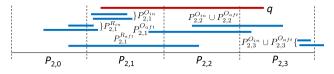

Fig. 8 Partition subdivisions in HINT^m (level $\ell = 2$)

$P_{\ell,i}^{O_{in}}$ that end *inside* (resp. *after*) partition $P_{\ell,i}$. Similarly, each $P_{\ell,i}^R$ is divided into $P_{\ell,i}^{R_{in}}$ and $P_{\ell,i}^{R_{aft}}$.

Queries that overlap with multiple partitions Consider a query q, which overlaps with a sequence of *more than one partitions* at level ℓ. As already discussed, if we have to conduct comparisons in the first such partition $P_{\ell,f}$, we should do so for all intervals in $P_{\ell,f}^O$ and $P_{\ell,f}^R$. By subdividing $P_{\ell,f}^O$ and $P_{\ell,f}^R$, we get the following lemma:

Lemma 5 *If $P_{\ell,f} \neq P_{\ell,l}$ (1) each interval s in $P_{\ell,f}^{O_{in}} \cup P_{\ell,f}^{R_{in}}$ overlaps with q iff $s.end \geq q.st$; and (2) all intervals s in $P_{\ell,f}^{O_{aft}}$ and $P_{\ell,f}^{R_{aft}}$ surely overlap with q.*

Proof Follows directly from the fact that q starts *inside* $P_{\ell,f}$ but ends *after* $P_{\ell,f}$. □

Hence, we need *just one comparison* for each interval in $P_{\ell,f}^{O_{in}} \cup P_{\ell,f}^{R_{in}}$, whereas we can report all intervals $P_{\ell,f}^{O_{aft}} \cup P_{\ell,f}^{R_{aft}}$ as query results *with no comparisons*. As already discussed, for all partitions $P_{\ell,i}$ between $P_{\ell,f}$ and $P_{\ell,l}$, we just report intervals in $P_{\ell,i}^{O_{in}} \cup P_{\ell,i}^{O_{aft}}$ as results, with no comparisons, whereas for the last partition $P_{\ell,l}$, we perform *one comparison* per interval in $P_{\ell,l}^{O_{in}} \cup P_{\ell,l}^{O_{aft}}$.

Queries that overlap with a single partition If the query q overlaps only one partition $P_{\ell,f}$ at level ℓ, we can use following lemma to minimize the necessary comparisons:

Lemma 6 *If $P_{\ell,f} = P_{\ell,l}$, then*

- *each interval s in $P_{\ell,f}^{O_{in}}$ overlaps with q iff $s.st \leq q.end \wedge q.st \leq s.end$,*
- *each interval s in $P_{\ell,f}^{O_{aft}}$ overlaps with q iff $s.st \leq q.end$,*
- *each interval s in $P_{\ell,f}^{R_{in}}$ overlaps with q iff $s.end \geq q.st$,*
- *all intervals in $P_{\ell,f}^{R_{aft}}$ overlap with q.*

Proof All intervals $s \in P_{\ell,f}^{O_{aft}}$ end after q, so $s.st \leq q.end$ suffices as an overlap test. All intervals $s \in P_{\ell,f}^{R_{in}}$ start before q, so $s.st \leq q.end$ suffices as an overlap test. All intervals $s \in P_{\ell,f}^{R_{aft}}$ start before and end after q, so they are guaranteed results. □

Overall, the subdivisions help us to minimize the number of intervals in each partition, for which we have to apply comparisons. Figure 8 shows the subdivisions which are accessed by query q at level $\ell = 2$ of a HINT^m index. In partition

Table 3 Necessary data and beneficial sort orders

Subdivision	Beneficial sorting	Necessary data
$P_{\ell,i}^{O_{in}}$	By $s.st$ or by $s.end$	$s.id, s.st, s.end$
$P_{\ell,i}^{O_{aft}}$	By $s.st$	$s.id, s.st$
$P_{\ell,i}^{R_{in}}$	By $s.end$	$s.id, s.end$
$P_{\ell,i}^{R_{aft}}$	No sorting	$s.id$

$P_{\ell,f} = P_{2,1}$, all four subdivisions are accessed, but comparisons are needed only for intervals in $P_{2,1}^{O_{in}}$ and $P_{2,1}^{R_{in}}$. In $P_{2,2}$, only the originals (in $P_{2,2}^{O_{in}}$ and $P_{2,2}^{O_{aft}}$) are accessed and reported without any comparisons. Finally, in $P_{\ell,l} = P_{2,3}$, only the originals (in $P_{2,3}^{O_{in}}$ and $P_{2,3}^{O_{aft}}$) are accessed and compared to q.

4.1.1 Sorting the intervals in each subdivision

We can keep the intervals in each subdivision sorted, in order to reduce the number of comparisons for queries that access them. For example, let us examine the last partition $P_{\ell,l}$ that overlaps with a query q at a level ℓ. If the intervals s in $P_{\ell,l}^{O_{in}}$ are sorted on their start endpoint (i.e., $s.st$), we can simply access and report the intervals until the first $s \in P_{\ell,l}^{O_{in}}$, such that $s.st > q.end$. Or, we can perform binary search to find the first $s \in P_{\ell,l}^{O_{in}}$, such that $s.st > q.end$ and then scan and report all intervals before s. Table 3 (second column) summarizes the sort orders for each of the four subdivisions of a partition that can be beneficial in query evaluation. For a subdivision $P_{\ell,i}^{O_{in}}$, intervals may have to be compared based on their start point (if $P_{\ell,i} = P_{\ell,f}$), or based on their end point (if $P_{\ell,i} = P_{\ell,l}$), or based on both points (if $P_{\ell,i} = P_{\ell,f} = P_{\ell,l}$). We choose to sort based on $s.st$ to accommodate two of these three cases. For a subdivision $P_{\ell,i}^{O_{aft}}$, intervals may have to be compared *only* based on their start point (if $P_{\ell,i} = P_{\ell,l}$). For a subdivision $P_{\ell,i}^{R_{in}}$, intervals may have to be compared *only* based on their end point (if $P_{\ell,i} = P_{\ell,f}$). Last, for a subdivision $P_{\ell,i}^{R_{aft}}$, there is never any need to compare the intervals, so, no order provides any benefit. Overall, sorting will reduce the expected number of comparisons per query for $P_{\ell,l}^{O_{aft}}$ and $P_{\ell,l}^{R_{in}}$ to $O(log(n/2^m))$, but the expected cost for $P_{\ell,l}^{O_{in}}$ remains $O(n/2^m)$. Under this, the worst-case query cost remains $O(n + K)$, where K is the number of query results, derived from Theorem 2.
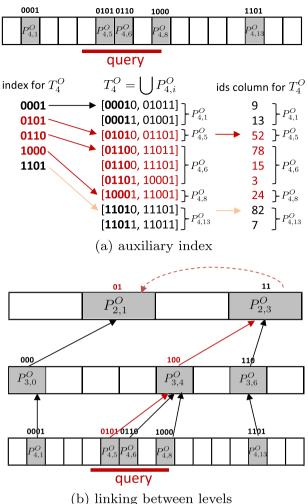
4.1.2 Storage optimization

So far, we have assumed that each interval s is stored in the partitions whereto s is assigned as a triplet $\langle s.id, s.st, s.end \rangle$. However, if we split the partitions into subdivisions, we do not need to keep all information of the intervals in them. Specifically, for each subdivision $P_{\ell,i}^{O_{in}}$, we may need to use $s.st$ and/or $s.end$ for each interval $s \in P_{\ell,i}^{O_{in}}$, while for each subdivision $P_{\ell,i}^{O_{aft}}$, we may need to use $s.st$ for each $s \in P_{\ell,i}^{O_{in}}$, but we will never need $s.end$. From the intervals s of each subdivision $P_{\ell,i}^{R_{in}}$, we may need $s.end$, but we will never use $s.st$. Finally, for each subdivision $P_{\ell,i}^{R_{aft}}$, we just have to keep the $s.id$ identifiers of the intervals. Table 3 (third column) summarizes the data that we need to keep from each interval in the subdivisions of each partition. Since each interval s is stored as original just once in the entire index, but as replica in possibly multiple partitions, space can be saved by storing only the necessary data, especially if the intervals span multiple partitions. Note that even when we do not apply the subdivisions, but just use $P_{\ell,i}^{O}$ and $P_{\ell,i}^{R}$ (as suggested in Sect. 3.2), we do not need to store the start points $s.st$ of all intervals in $P_{\ell,i}^{R}$, as they are never used in comparisons.

4.2 Handling data skewness and sparsity

Data skewness and sparsity may cause many partitions to be empty, especially at the lowest levels of HINT (i.e., large values of ℓ). Recall that a query accesses a sequence of multiple $P_{\ell,i}^{O}$ partitions at each level ℓ. Since the intervals are physically distributed in the partitions, this results into the unnecessary accessing of empty partitions and may cause cache misses. We propose a storage organization where all $P_{\ell,i}^{O}$ divisions at the same level ℓ are merged into a single table T_ℓ^{O} and an auxiliary index is used to find each non-empty division. [3] The auxiliary index locates the first non-empty partition, which is greater than or equal to the ℓ-prefix of $q.st$ (i.e., via binary search or a binary search tree). From thereon, the nonempty partitions which overlap with the query interval are accessed sequentially and distinguished with the help of the auxiliary index. Hence, the contents of the relevant $P_{\ell,i}^{O}$'s to each query are always accessed sequentially. Figure 9(a) shows an example at level $\ell = 4$ of HINT^m. From the total $2^\ell = 16$ P^O partitions at that level, only 5 are nonempty (shown in gray at the top of the figure): $P_{4,1}^{O}, P_{4,5}^{O}, P_{4,6}^{O}, P_{4,8}^{O}, P_{4,13}^{O}$. All 9 intervals in them (sorted by start point) are unified in a single table T_4^{O} as shown at the bottom of the figure (the binary representations of the interval endpoints are shown). At the moment, ignore the ids column for T_4^{O} at the right of the figure. The sparse index for T_4^{O} has one entry per nonempty partition pointing to the first interval in it. For the query in the example, the index is used to find the first nonempty partition $P_{4,5}^{O}$, for

---

[3] For simplicity, we discuss this organization when a partition $P_{\ell,i}$ is divided into $P_{\ell,i}^{O}$ and $P_{\ell,i}^{R}$; the same idea can be straightforwardly applied also when the four subdivisions discussed in Sect. 4.1 are used.

(a) auxiliary index

(b) linking between levels

Fig. 9 Storage and indexing optimizations

which the id is greater than or equal to the 4-bit prefix 0100 of $q.st$. All relevant non-empty partitions $P_{4,5}^O$, $P_{4,6}^O$, $P_{4,8}^O$ are accessed sequentially from T_4^O, until the position of the first interval of $P_{4,13}^O$.

Searching for the first partition $P_{\ell,f}^O$ that overlaps with q at each level can be quite expensive when numerous nonempty partitions exist. To alleviate this issue, we add to the auxiliary index, a link from each partition $P_{\ell,i}^O$ to the partition $P_{\ell-1,j}^O$ at the level above, such that j is the smallest number greater than or equal to $i \div 2$, for which partition $P_{\ell-1,j}^O$ is not empty. Hence, instead of performing binary search at level $\ell - 1$, we use the link from the first partition $P_{\ell,f}^O$ relevant to the query at level ℓ and (if necessary) apply a linear search backwards from the pointed partition $P_{\ell-1,j}^O$ to identify the first non-empty partition $P_{\ell-1,f}^O$ that overlaps with q. Figure 9(b) shows an example, where each nonempty partition at level ℓ is linked with the first nonempty partition with greater than or equal prefix at the level $\ell-1$ above. Given query example q, we use the auxiliary index to find the first nonempty parti-

tion $P_{4,5}^O$ which overlaps with q and also sequentially access $P_{4,6}^O$ and $P_{4,8}^O$. Then, we follow the pointer from $P_{4,5}^O$ to $P_{3,4}^O$ to find the first nonempty partition at level 3, which overlaps with q. We repeat this to get partition $P_{2,3}^O$ at level 2, which, however, is not guaranteed to be the first one overlapping with q, so we go backwards to $P_{2,3}^O$.

4.3 Reducing cache misses

At most levels of HINT^m, no comparisons are conducted and the only operations are processing the interval ids which qualify the query. Also, even for the levels ℓ where comparisons are required, these are only restricted to the first and the last relevant partitions $P_{\ell,f}^O$ and $P_{\ell,l}^O$ and no comparisons are needed for the partitions in-between. Summing up, when accessing any (sub-)partition for which no comparison is required, we do not need any information about the intervals, except for their ids. Hence, in our implementation, for each (sub-)partition, we store the ids of all intervals in it in a dedicated array (the *ids column*) and the interval endpoints (wherever necessary) in a different array.[4] If we need the id of an interval that qualifies a comparison, we can access the corresponding position of the ids column. This storage organization greatly improves search performance by reducing the cache misses, because for the intervals that do not require comparisons, we only access their ids and not their interval endpoints. This optimization is orthogonal to and applied in combination with the strategy in Sect. 4.2, i.e., we store all P^O divisions at each level ℓ in a single table T_ℓ^O, which is decomposed to a column that stores the ids and another table for the endpoint data of the intervals. We exemplify the ids column in Fig. 9(a). If, for a sequence of partitions at a level, we do not have to perform any comparisons, we just access the sequence of the interval ids that are part of the answer, which is implied by the position of the first such partition (obtained via the auxiliary index). In this example, all intervals in $P_{4,5}^O$ and $P_{4,6}^O$ are guaranteed to be query results without any comparisons and they can be sequentially accessed from the ids column without having to access the endpoints of the intervals. The auxiliary index guides the search by identifying and distinguishing between partitions for which comparisons should be conducted (e.g., $P_{4,8}^O$) and those for which they are not necessary.

4.4 Updates

A version of HINT^m that uses *all* techniques from Sects. 4.1, 4.1.1, 4.1.2, 4.2, is optimized for query operations. Under this,

---

[4] Similar to the previous section, this storage optimization can be straightforwardly employed also when a partition is divided into $P_{\ell,i}^{O_{in}}$, $P_{\ell,i}^{O_{aft}}$, $P_{\ell,i}^{R_{in}}$, $P_{\ell,i}^{R_{aft}}$.

Table 4 Characteristics of real datasets

	BOOKS	WEBKIT	TAXIS	GREEND
Cardinality	2,312,602	2,347,346	172,668,003	110,115,441
Size (MBs)	27.8	28.2	2072	1321
Domain (s)	31,507,200	461,829,284	31,768,287	283,356,410
Min. duration (s)	1	1	1	1
Max. duration (s)	31,406,400	461,815,512	2,148,385	59,468,008
Avg. duration (s)	2,201,320	33,206,300	758	15
Avg. duration (%)	6.98	7.19	0.0024	0.000005

the index cannot efficiently support individual updates, i.e., new intervals inserted one-by-one. Dealing with updates in *batches* will be a better fit. This is a common practice for other update-unfriendly indices, e.g., the inverted index in IR. Yet, for mixed workloads (i.e., with both queries and updates), we adopt a hybrid setting where a *delta* index is maintained to digest the latest updates as discussed in Sect. 3.4,[5] and a fully optimized HINT^m, which is periodically updated in batches, holds older data supporting deletions with tombstones. Both indices are probed upon querying.

5 Experimental analysis

We compare our hierarchical indexing, detailed in Sects. 3 and 4 against the interval tree [18][6], the timeline index [21], the (adaptive) period index [4], and a uniform 1D-grid. All indices were implemented in C++ and compiled using gcc (v4.8.5) with -O3.[7] The tests ran on a dual Intel(R) Xeon(R) CPU E5-2630 v4 at 2.20GHz with 384 GBs of RAM, running CentOS Linux.

5.1 Data and queries

We used 4 collections of real intervals, which have also been used in previous works; Table 4 summarizes their characteristics. BOOKS [8] contains the periods of time in 2013 when books were lent out by Aarhus libraries (https://www.odaa.dk). WEBKIT [8, 9, 15, 33] records the file history in the git repository of the Webkit project from 2001 to 2016 (https://webkit.org); the intervals indicate the periods during which a file did not change. TAXIS [10] stores the time periods of taxi trips (pick-up and drop-off timestamps) from NY City in 2013 (https://www1.nyc.gov/site/tlc/index.page). GREEND [11, 28] records time periods of power usage from households in Austria and Italy from January 2010 to October

Table 5 Parameters of synthetic datasets

Parameter	Values (Defaults in **bold**)
Domain length	32 M, 64 M, **128 M**, 256 M, 512 M
Cardinality	**10 M**, 50 M, 100 M, 500 M, 1 B
α (Interval length)	1.01, 1.1, **1.2**, 1.4, 1.8
σ (Interval position)	10 K, 100 K, **1 M**, 5 M, 10 M

2014. BOOKS and WEBKIT contain around 2 M intervals each, which are quite long on average; TAXIS and GREEND have over 100 M short intervals.

We also generated synthetic collections to simulate different cases for the lengths and the skewness of the input intervals. Table 5 shows the construction parameters for the synthetic datasets and their default values. The domain of the datasets ranges from 32 M to 512 M, which requires index level parameter m to range from 25 to 29 for a comparison-free HINT (similar to the real datasets). The cardinality ranges from 10 M to 1B. The interval lengths were generated using the random.zipf(α) in the numpy library. They follow a zipfian distribution according to the $p(x) = \frac{x^{-a}}{\zeta(a)}$ probability density function, where ζ is the Riemann Zeta function. A small value of α results in most intervals being relatively long, while a large value results in the great majority of intervals having length 1. We generated the positions of the *middle points* of the intervals from a normal distribution centered at the middle point μ of the domain. So, the middle point of each interval is generated using numpy's random.normalvariate(μ, σ). The greater the value of σ, the more spread the intervals are in the domain.

On the real datasets, we used queries uniformly distributed in the domain. On the synthetic, the query positions follow the distribution of the data. In both, the query extent was fixed to a percentage of the domain size (default 0.1%). At each test, we ran 10K random queries to measure the overall throughput. Measuring query throughput instead of average query time makes sense in applications that manage huge volumes of interval data and offer a search interface to billions of users simultaneously (e.g., public historical databases).

---

[5] Small adjustments are applied for the $P_{l,i}^{O_{in}}$, $P_{l,i}^{O_{aft}}$, $P_{l,i}^{R_{in}}$, $P_{l,i}^{R_{aft}}$ subdivisions and the storage optimizations.

[6] Code from https://github.com/ekg/intervaltree.
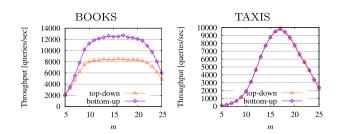
[7] Source code available in https://github.com/pbour/hint.

Fig. 10 Optimizing HINT^m: query evaluation approaches

5.2 Optimizing HINT/HINT^m

In our first set of tests, we study the best setting for our hierarchical indexing. We compare the effectiveness of the two evaluation approaches in Sect. 3.2.1 and investigate the impact of the optimizations in Sect. 4.

5.2.1 Query evaluation approaches on HINT^m

We compare the straightforward *top-down* approach for evaluating queries on HINT^m that uses solely Lemma 1, against the *bottom-up* illustrated in Algorithm 3 which additionally employs Lemma 2. Figure 10 reports the throughput of each approach on BOOKS and TAXIS, while varying the number of levels m in the index. We omit the results for WEBKIT and GREEND that follow identical trend to BOOKS and TAXIS, respectively. We observe that the *bottom-up* approach significantly outperforms *top-down* for BOOKS while for TAXIS, this performance gap is very small. As expected, *bottom-up* performs at its best for inputs that contain long intervals which are indexed on high levels of the index, i.e., the intervals in BOOKS. In contrast, the intervals in TAXIS are very short and so, indexed at the bottom level of HINT^m, while the majority of the partitions at the higher levels are empty. Hence, *top-down* conducts no comparisons at higher levels. For the rest of our tests, HINT^m uses the *bottom-up* approach.

5.2.2 Subdivisions and space decomposition

We next evaluate the *subdivisions* and *space decomposition* optimizations described in Sect. 4.1 for HINT^m. Note that these techniques are not applicable to our comparison-free HINT as the index stores only interval ids. Figure 11 shows the effect of the optimizations on BOOKS and TAXIS, for different values of m; similar trends were observed in WEBKIT and GREEND, respectively. The plots include (1) a *base* version of HINT^m, which employs none of the proposed optimizations, (2) *subs+sort+opt*, with all optimizations activated, (3) *subs+sort*, which only sorts the subdivisions (Sect. 4.1.1) and (4) *subs+sopt*, which uses only the storage optimization for the subdivisions (Sect. 4.1.2). We observe that the *subs+sort+opt* version of HINT^m is superior
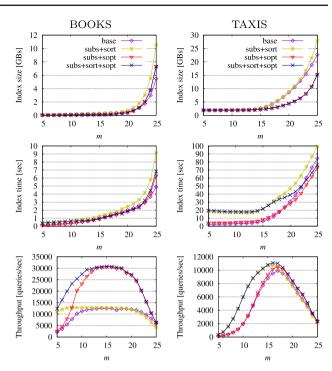

Fig. 11 Optimizing HINT^m: subdivisions and space decomposition

Table 6 Optimizing HINT: impact of the skewness & sparsity optimization (Sect. 4.2), default parameters

Dataset	Throughput (queries/s)		Index size (MBs)	
	Original	Optimized	Original	Optimized
BOOKS	12098	36173	3282	273
WEBKIT	947	39000	49439	337
TAXIS	2931	31027	10093	7733
GREEND	648	47038	57667	10131

to all three other versions, on all tests. Essentially, the index benefits from the *sub+sort* setting only when m is small, i.e., below 15, at the expense of increasing the index time compared to *base*. In this case, the partitions contain a large number of intervals and therefore, using binary search or scanning until the first interval that does not overlap the query, will save on the conducted comparisons. On the other hand, the *subs+sopt* optimization significantly reduces the space requirements of the index. As a result, the version incurs a higher cache hit ratio and so, a higher throughput compared to *base* is achieved, especially for large values of m, i.e., higher than 10. The *subs+sort+opt* version manages to combine the benefits of both *subs+sort* and *subs+sopt* versions, i.e., high throughput in all cases, with low space requirements. The effect in the performance is more pronounced in BOOKS because of the long intervals and the high replication ratio. In view of these results, HINT^m employs all optimizations from Sect. 4.1 for the rest of our experiments.
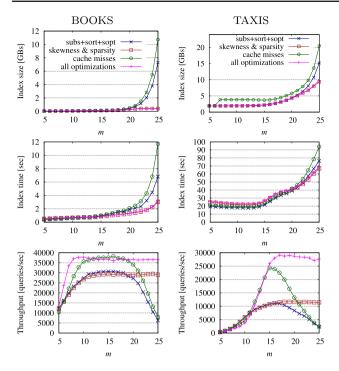
Fig. 12 Optimizing HINT^m: impact of handling skewness & sparsity and reducing cache misses optimizations

5.2.3 Handling data skewness & sparsity and reducing cache misses

Table 6 tests the effect of the *handling data skewness & sparsity* optimization (Sect. 4.2) on the comparison-free version of HINT (Sect. 3.1).[8] Observe that the optimization has a great effect on both the throughput and the size of the index in all four real datasets, because empty partitions are effectively excluded from query evaluation and from the indexing process.

Figure 12 shows the effect of either or both of the *data skewness & sparsity* (Sect. 4.2) and the *cache misses* optimizations (Sect. 4.3) on the performance of HINT^m for different values of m. In all cases, the version of HINT^m which uses both optimizations is superior to all other versions. As expected, the *skewness & sparsity* optimization helps to reduce the space requirements of the index when m is large, because there are many empty partitions in this case at the bottom levels of the index. At the same time, the *cache misses* optimization helps in reducing the number of cache misses in all cases where no comparisons are needed. Overall, the optimized version of $HINT^m$ converges to its best performance at a relatively small value of m, where the space requirements of the index are relatively low, especially

---

[8] The *cache misses* optimization (Sect. 4.3) is only applicable to HINT^m.

on the BOOKS and WEBKIT datasets which contain long intervals.

For the rest of our experiments, HINT^m employs both optimizations and HINT the *data skewness & sparsity* optimization. Last, by juxtaposing Table 7 with Figs. 11 and 12, we also observe that both m_{opt} values correspond to the part of the plots before the index size blows, usually for $m \geq 20$.

5.2.4 Tuning m

After demonstrating the merit of HINT^m optimizations, we now elaborate on how to set the value of m and on the effectiveness of our analytical model from Sect. 3.3. As we already discussed our model is based on the intuition that as m increases, the cost of accessing comparison-free results dominates the computational cost of the comparisons. Figure 13 confirms our intuition on BOOKS and TAXIS (the plots for WEBKIT and GREEND exhibit exactly the same trend as BOOKS and TAXIS, respectively). For different values of m and for $10K$ queries, we report the overall time spend for comparisons between data intervals and query intervals, denoted by C_{cmp}, and the overall time spent to output results with no comparisons, denoted by C_{acc}, i.e., the time taken for simply accessing data intervals which are guaranteed query results. We also include the total execution time, i.e., $C_{cmp} + C_{acc}$.

The plots clearly show the expected behavior. For small values of m, the cost of conducting comparisons dominates the total execution cost since the partitions at the bottom level m of the index have large extents and numerous intervals. As m increases, the fraction of the results collected from just accessing the contents of partitions rises, increasing the C_{acc} cost. The optimal values m_{opt} (i.e., where the total execution time is the lowest possible) occur after C_{acc} exceeds C_{cmp}. In fact, we notice that increasing m beyond m_{opt} roughly eliminates the cost of comparisons ($C_{cmp} \approx 0$) as the partitions are much shorter than the queries, while the total cost essentially equals the cost of simply accessing the intervals from the comparison-free partitions.

To determine m_{opt}, our model in Sect. 3.3 selects the smallest m value for which the index converges within 3% to its lowest estimated cost. Table 7 reports, for each real dataset, m_{opt} (est.) and m_{opt} (exps), which brings the highest throughput in our tests. Overall, our model estimates a value of m_{opt} which is very close to m_{opt} (exps). Despite a larger gap for WEBKIT, the measured throughput for the estimated $m_{opt} = 9$ is only 5% lower than the best observed throughput.

5.2.5 Discussion

Table 7 also shows the *replication factor k* of the index, i.e., the average number of partitions in which every interval is

Table 7 Statistics and parameter setting

Index	Parameter	BOOKS	WEBKIT	TAXIS	GREEND
Period	#levels	4	4	7	8
	#coarse partitions	100	100	100	100
Timeline	#Checkpoints	6000	6000	8000	8000
1D-grid	#Partitions	500	300	4000	30000
HINT^m	m_{opt} (est.)	9	9	16	16
	m_{opt} (exps)	10	12	17	17
	Rep. factor k (est.)	6.09	8.98	1.98	1
	Rep. factor k (exps)	5.13	6.07	2.14	1.0013
	Avg. comp. part.	3.226	3.538	3.856	2.937
	No comp. results	99.9%	99.9%	99.8%	99.3%

Fig. 13 Setting m: measured costs

Table 8 Comparing index size (MBs)

Index	BOOKS	WEBKIT	TAXIS	GREEND
Interval tree	97	115	3125	2241
Period	210	217	2278	**1262**
Timeline	4916	5671	4203	2525
1D-grid	949	604	2165	1264
HINT	273	337	7733	10131
HINT^m	**81**	**98**	**2039**	1278

Table 9 Comparing index time (s)

Index	BOOKS	WEBKIT	TAXIS	GREEND
Interval tree	**0.25**	**0.33**	47.2	26.8
Period	1.15	1.35	76.9	46.4
Timeline	12.7	19.2	40.4	15.9
1D-grid	1.26	0.95	**4.02**	**2.24**
HINT	1.70	11.8	49.6	36.5
HINT^m	0.73	0.53	22.8	8.58

stored, as predicted by our space complexity analysis (see Theorem 1) and as measured experimentally. As expected, the replication factor is high on BOOKS, WEBKIT due to the large number of long intervals, and low on TAXIS, GREEND where the intervals are very short and stored at the bottom levels. Although our analysis uses simple statistics, the predictions are quite accurate.

The next line of the table (*avg. comp. part.*) shows the average number of HINT^m partitions for which comparisons were conducted. Consistently to our analysis in Sect. 3.2.3, all numbers are below 4, which means that the performance of HINT^m is very close to the performance of the comparison-free, but space-demanding HINT. To further elaborate on the number of required comparisons, we last show the fraction of the results produced by HINT^m without any comparisons. In all datasets, over 99% of the results are collected with no comparisons, which explains how HINT^m is able to match the performance of the comparison-free HINT.

5.3 Index performance comparison

Next, we compare the optimized versions of HINT and HINT^m against the previous work competitors. We start with our tests on the real datasets. For HINT^m, we set m to the best value on each dataset, according to Table 7. Similarly, we set the number of partitions for 1D-grid, the number of checkpoints for the timeline index, and the number of levels and number of coarse partitions for the period index (see

Table 7). Table 8 shows the sizes of each index in memory and Table 9 shows the construction cost of each index, for the default query extent 0.1%. Regarding space, HINT^m along with the interval tree and the period index have the lowest requirements on datasets with long intervals (BOOKS and WEBKIT) and very similar to 1D-grid in the rest. In TAXIS and GREEND where the intervals are indexed mainly at the bottom level, the space requirements of HINT^m are significantly lower than our comparison-free HINT due to limiting the number of levels. When compared to the raw data (see Table 4), HINT^m is 2 to 3 times bigger for BOOKS and WEBKIT (which contain many long intervals), and 1 time bigger for GREEND and TAXIS. These ratios are smaller than the replication ratios k reported in Table 7, thanks to our storage optimization (cf. Section 4.1.2). Due to its simplicity, 1D-grid has the lowest index time across all datasets.
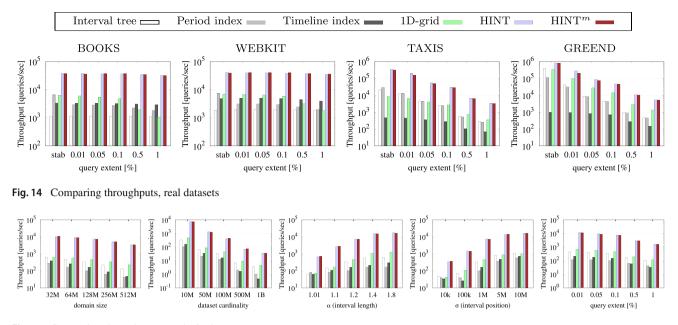
Fig. 14 Comparing throughputs, real datasets

Fig. 15 Comparing throughputs, synthetic datasets

Nevertheless, HINT^m is the runner up in most of the cases, especially for the biggest inputs, i.e., TAXIS and GREEND, while in BOOKS and WEBKIT, its index time is very close to the interval tree.

Figure 14 compares the throughputs of all indices on queries of various extents (as a percentage of the domain size). The first set of bars in each plot corresponds to *stabbing* queries, i.e., queries of 0 extent. We observe that HINT and HINT^m outperform the competition by almost one order of magnitude, across the board. In fact, only on GREEND the performance for one of the competitors, i.e., 1D-grid, comes close to the performance of our hierarchical indexing. Due to the extremely short intervals in GREEND (see Table 4) almost all the results are collected from the bottom level of HINT/HINT^m, which essentially resembles the evaluation process in 1D-grid. Yet, our indices are even in this case faster as they require no duplicate elimination.

HINT^m is the best index overall, as it achieves the performance of HINT, requiring less space, confirming the findings of our analysis in Sect. 3.2.3. As shown in Table 8, HINT always has higher space requirements than HINT^m; even up to an order of magnitude higher in case of GREEND. What is more, since HINT^m offers the option to control the occupied space in memory by appropriately setting the m parameter, it can handle scenarios with space limitations. HINT is marginally better than HINT^m only on datasets with short intervals (TAXIS and GREEND) and only for selective queries. In these cases, the intervals are stored at the lowest levels of the hierarchy where HINT^m typically needs to conduct comparisons to identify results, but HINT applies comparison-free retrieval.

We next consider the synthetic datasets. In each test, we vary the value of one parameter (domain size, cardinality, α, σ, query extent) and fix the rest to their default (see Table 5). The value of m for HINT^m, the number of partitions for 1D-grid, the number of checkpoints for the timeline index and the number of levels/coarse partitions for the period index are set to their best values on each dataset. The results from Fig. 15 follow a similar trend to the tests on the real datasets. HINT and HINT^m are always significantly faster than the competition.

Different to the real datasets, 1D-grid is steadily outperformed by the other three competitors. Intuitively, the uniform partitioning of the domain in 1D-grid cannot cope with the skewness of the synthetic datasets. As expected the domain size, the dataset cardinality and the query extent have a negative impact on all indices. Essentially, increasing the domain size under a fixed query extent affects the performance similar to increasing the query extent, i.e., the queries become longer and less selective, including more results. Further, the querying cost grows linearly with the dataset size since the number of query results is proportional to it. HINT^m occupies around 8% more space than the raw data, because the replication factor k is close to 1. In contrast, as α grows, the intervals become shorter, so the query performance improves. Similarly, when increasing σ the intervals are more widespread, meaning that the queries are expected to retrieve fewer results, and the query cost drops accordingly.

Table 10 Throughput (operations/s), total cost (s)

Index	Operation			Total cost
	Queries	Insertions	Deletions	
BOOKS				
Interval tree	1258	5841	1142	9.63
Period index	3088	519,904	765	4.52
1D-grid	3739	411,540	165	8.68
$_{subs+sopt}$HINT^m	14,390	2,405,228	2201	1.14
HINT^m	**40,311**	**3,680,457**	**5928**	**0.41**
TAXIS				
Interval tree	2619	61,923	14,318	3.93
Period index	2695	1,026,423	21,293	3.76
1D-grid	2572	**8,347,273**	16,236	3.95
$_{subs+sopt}$HINT^m	8774	4,407,743	71,122	1.15
HINT^m	**28,596**	6,745,622	**90,460**	**0.36**

5.4 Updates

We now test the efficiency of HINT^m in updates using both the update-friendly version of HINT^m (Sect. 3.4), denoted by $_{subs+sopt}$HINT^m, and the hybrid setting for the fully optimized index from Sect. 4.4, denoted as HINT^m. We index offline the first 90% of the intervals for each real dataset in batch and then execute a mixed workload with 10K queries of 0.1% extent, 5K insertions of new intervals (randomly selected from the remaining 10% of the dataset) and 1K random deletions. Table 10 reports our findings for BOOKS and TAXIS; the results for WEBKIT and GREEND follow the same trend. Note that we excluded Timeline since the index is designed for temporal (versioned) data where updates only happen as new events are appended at the end of the event list, and the comparison-free HINT, for which our tests have already shown a similar performance to HINT^m with higher indexing/ storing costs. Also, all indices handle deletions with "tombstones." We observe that both versions of HINT^m outperform the competition by a wide margin. An exception arises on TAXIS, as the short intervals are inserted in only one partition in 1D-grid. The interval tree has in fact several orders of magnitude slower updates due to the extra cost of maintaining the partitions in the tree sorted at all time. Over-

all, we also observe that the hybrid HINT^m setting is the most efficient index as the *smaller* delta $_{subs+sopt}$HINT^m handles insertions faster than the 90% pre-filled $_{subs+sopt}$HINT^m.

5.4.1 Interval joins

We conclude the first part of our analysis studying the applicability of HINT^m to the evaluation of interval joins. Given two input datasets R, S, the objective is to find all pairs of intervals $(r, s), r \in R, s \in S$, such that r G-OVERLAPS with s. The rationale is that if the outer dataset R is very small compared to the inner S, an index *already* available for S can be used to evaluate fast the join in an index nested-loops fashion. Hence, we show how HINT^m constructed for each of the four real datasets can be used to evaluate joins where the outer relation is a random sample of the same dataset. As part of the join process, we sort the outer dataset R in order to achieve better cache locality between consecutive probes to the inner dataset S. As a competitor, we used the state-of-the-art interval join algorithm [10], which sorts both join inputs and applies a specialized sweeping algorithm optFS. Figure 16 shows the results for various sizes $|R|$ of the outer dataset R. The results confirm our expectation. For small sizes of $|R|$, HINT^m is able to outperform optFS. On TAXIS in particular, HINT^m loses to [10] only when $|R|/|S| \geq 50\%$.

6 Supporting Allen's algebra

We now turn our focus to Allen's algebra for intervals [1]. Table 11 (first two columns) summarizes the basic relationships of the algebra, each denoted by q REL s, where q is the query interval and s, an interval in the input collection S. Note that the G-OVERLAPS selection query from the previous sections identifies every interval s non-disjoint to query q, i.e., a combination of all basic algebra's relationships besides BEFORE and AFTER.

We study selection queries on Allen's relationships under two setups for our hierarchical indexing. We focus on HINT^m, which exhibits similar performance to the comparison-free HINT but significant lower indexing costs, as our experiments showed in Sect. 5.
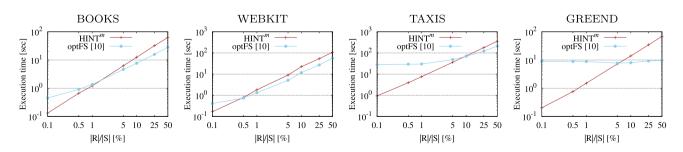

Fig. 16 G-OVERLAPS based interval joins, real datasets

Table 11 Supporting Allen's algebra, setup optimized for G-OVERLAPS relationship (Table 3)

q REL s	Definition	Result set
EQUALS	$q.st = s.st \wedge$ $q.end = s.end$	**if** $f = l,$ $\left\{ s \in P_{\ell,f}^{O_{in}} : q.st = s.st \wedge q.end = s.end \right\}$ $\textbf{else} \left\{ s \in P_{\ell,f}^{O_{aft}} : q.st = s.st \right\} \bigcap \left\{ s \in P_{\ell,l}^{R_{in}} : q.end = s.end \right\}$
STARTS	$q.st = s.st \wedge$ $q.end < s.end$	$\forall \ell: \textbf{if } f = l, \left\{ s \in P_{\ell,f}^{O_{in}} : q.st = s.st \wedge q.end < s.end \right\} \bigcup \left\{ s \in P_{\ell,f}^{O_{aft}} : q.st = s.st \right\}$ $\textbf{else} \left\{ s \in P_{\ell,f}^{O_{aft}} : q.st = s.st \right\} \bigcap \left\{ \bigcup_{\forall \ell'} \left\{ s \in P_{\ell',l}^{R_{in}} : q.end < s.end \right\} \bigcup P_{\ell,l}^{R_{aft}} \right\}$
STARTED_BY	$q.st = s.st \wedge$ $q.end > s.end$	$\forall \ell: \textbf{if } f = l, \left\{ s \in P_{\ell,f}^{O_{in}} : q.st = s.st \wedge q.end > s.end \right\}$ $\textbf{else} \left\{ s \in P_{\ell,f}^{O_{in}} : q.st = s.st \right\} \bigcup$ $\left\{ \left\{ s \in P_{\ell,f}^{O_{aft}} : q.st = s.st \right\} \bigcap \left\{ \bigcup_{\forall \ell'} \left\{ \bigcup_{\forall f < i < l} P_{\ell',i}^{R_{in}} \right\} \bigcup \left\{ s \in P_{\ell,l}^{R_{in}} : q.end > s.end \right\} \right\} \right\}$
FINISHES	$q.end = s.end \wedge$ $q.st > s.st$	$\forall \ell: \textbf{if } f = l, \left\{ s \in P_{\ell,l}^{O_{in}} : q.end = s.end \wedge q.st > s.st \right\} \bigcup \left\{ s \in P_{\ell,l}^{R_{in}} : q.end = s.end \right\}$ $\textbf{else} \left\{ s \in P_{\ell,l}^{R_{in}} : q.end = s.end \right\} \bigcap \left\{ \bigcup_{\forall \ell'} \left\{ s \in P_{\ell,f}^{O_{aft}} : q.st > s.st \right\} \bigcup P_{\ell,f}^{R_{aft}} \right\}$
FINISHED_BY	$q.end = s.end \wedge$ $q.st < s.st$	$\forall \ell: \textbf{if } f = l, \left\{ s \in P_{\ell,l}^{O_{in}} : q.end = s.end \wedge q.st < s.st \right\}$ $\textbf{else} \left\{ s \in P_{\ell,l}^{O_{in}} : q.end = s.end \right\} \bigcup$ $\left\{ \left\{ s \in P_{\ell,l}^{R_{in}} : q.end = s.end \right\} \bigcap \left\{ \bigcup_{\forall \ell'} \left\{ \left\{ s \in P_{\ell',f}^{O_{aft}} : q.st < s.st \right\} \bigcup \left\{ \bigcup_{\forall f < i < l} P_{\ell',i}^{O_{aft}} \right\} \right\} \right\} \right\}$
MEETS	$q.end = s.st$	$\forall \ell: \left\{ s \in P_{\ell,l}^{O_{in}} \bigcup P_{\ell,l}^{O_{aft}} : q.end = s.st \right\}$
MET_BY	$q.st = s.end$	$\forall \ell: \left\{ s \in P_{\ell,f}^{O_{in}} \bigcup P_{\ell,f}^{R_{in}} : q.st = s.end \right\}$
OVERLAPS	$q.st < s.st \wedge$ $q.end > s.st \wedge$ $q.end < s.end$	$\forall \ell: \textbf{if } f = l, \left\{ s \in P_{\ell,f}^{O_{in}} : q.st < s.st \wedge q.end > s.st \wedge q.end < s.end \right\} \bigcup$ $\left\{ s \in P_{\ell,f}^{O_{aft}} : q.st < s.st \wedge q.end > s.st \right\}$ $\textbf{else} \left\{ s \in P_{\ell,f}^{O_{in}} : q.end > s.st \wedge q.end < s.end \right\} \bigcup \left\{ s \in P_{\ell,l}^{O_{aft}} : q.end > s.st \right\}$ $\left\{ \left\{ s \in P_{\ell,l}^{R_{in}} : q.end < s.end \right\} \bigcup P_{\ell,l}^{R_{aft}} \right\} \bigcap \left\{ \bigcup_{\forall \ell'} \left\{ \left\{ s \in P_{\ell',f}^{O_{aft}} : q.st < s.st \right\} \bigcup \left\{ \bigcup_{\forall f < i < l} P_{\ell',i}^{O_{aft}} \right\} \right\} \right\}$
OVERLAPPED_BY	$q.st > s.st \wedge$ $q.st < s.end \wedge$	$\forall \ell: \textbf{if } f = l, \left\{ s \in P_{\ell,f}^{O_{in}} : q.st > s.st \wedge q.st < s.end \wedge q.end > s.end \right\}$ $\left\{ s \in P_{\ell,f}^{R_{in}} : q.st < s.end \wedge q.end > s.end \right\}$

Table 11 continued

q REL s	Definition	Result set
	$q.end > s.end$	$\textbf{else } \{s \in P_{\ell,f}^{O_{in}}: q.st > s.st \wedge q.st < s.end\} \cup \{s \in P_{\ell,f}^{R_{in}}: q.st < s.end\} \cup$ $\{\{\{s \in P_{\ell,f}^{O_{aft}}: q.st > s.st\} \cup P_{\ell,f}^{R_{aft}}\} \cap \{\bigcup_{\forall f<i<l}\{s \in P_{\ell',i}^{R_{in}}: q.end > s.end\} \cup \{s \in P_{\ell',l}^{R_{in}}: q.end > s.end\}\}\}$
CONTAINS	$q.st < s.st \wedge$ $q.end > s.end$	$\forall \ell: \textbf{if } f = l, \{s \in P_{\ell,f}^{O_{in}}: q.st < s.st \wedge q.end > s.end\}$ $\textbf{else } \{s \in P_{\ell,f}^{O_{in}}: q.st < s.st\} \cup \{\bigcup_{\forall f<i<l} P_{\ell,i}^{O_{in}}\} \cup \{\bigcup_{\forall f<i<l} P_{\ell,i}^{O_{aft}}\} \cap \{\bigcup_{\forall \ell'} \{\{s \in P_{\ell',l}^{R_{in}}: q.end > s.end\} \cup \{s \in P_{\ell',l}^{R_{in}}: q.end > s.end\}\}$
CONTAINED_BY	$q.st > s.st \wedge$ $q.end < s.end$	$\forall \ell: \textbf{if } f = l, \{s \in P_{\ell,f}^{O_{in}}: q.st > s.st \wedge q.end < s.end\} \cup P_{\ell,f}^{R_{aft}}$ $\{s \in P_{\ell,f}^{R_{in}}: q.end < s.end\} \cup P_{\ell,f}^{R_{aft}}$ $\textbf{else } \{\{s \in P_{\ell,f}^{O_{aft}}: q.st > s.st\} \cup P_{\ell,f}^{R_{aft}}\} \cap \{\bigcup_{\forall \ell'} \{\{s \in P_{\ell',l}^{R_{in}}: q.end < s.end\} \cup \{\bigcup_{\forall i>l}\{P_{\ell,i}^{O_{in}} \cup P_{\ell,i}^{O_{aft}}\}\}$
BEFORE	$q.end < s.st$	$\forall \ell: \{s \in P_{\ell,l}^{O_{in}} \cup P_{\ell,l}^{O_{aft}}: q.end < s.st\}$
AFTER	$q.st > s.end$	$\forall \ell: \{s \in P_{\ell,f}^{O_{in}} \cup P_{\ell,f}^{R_{in}}: q.st > s.end\} \cup \{\bigcup_{\forall i<f}\{P_{\ell,i}^{O_{in}} \cup P_{\ell,i}^{R_{in}}\}\}$

6.1 Setup optimized for G-OVERLAPS

We start off with the HINT^m setup from the first part of our paper (see Table 3), optimized for the G-OVERLAPS selection. In what follows, we discuss how queries based on Allen's relationships in Allen's algebra can be evaluated without any structural changes to the index. Table 11 summarizes the set of intervals reported for each selection query.

Relationship EQUALS An EQUALS selection determines all input intervals identical to query q, i.e., with $q.end = s.end$ and $q.st = s.st$. To answer such a query, we access two specific index partitions; the first relevant $P_{\ell,f}$ at level ℓ and the last relevant $P_{\ell',l}$, at level ℓ'.[9] Intuitively, these two partitions correspond to the first and last partition where HINT^m would store the query interval q, respectively. We then distinguish between two cases. If q overlaps a single partition, i.e., if $f = l$, we need only the intervals that both *start* and *end* inside this partition, i.e., the $P_{\ell,f}^{O_{in}}$ subdivision. So, we report set $\{s \in P_{\ell,f}^{O_{in}}: q.st = s.st \wedge q.end = s.end\}$. Otherwise, if $f \neq l$, we report results among the intervals that *start* in the first relevant partition (from $P_{\ell,f}^{O_{aft}}$) and *end* in the last (from $P_{\ell',l}^{R_{in}}$), i.e., set $\{s \in P_{\ell,f}^{O_{aft}}: q.st = s.st\} \cap \{s \in P_{\ell',l}^{R_{in}}: q.end = s.end\}$. Note that we cannot directly check $q.end = s.end$ as $P_{\ell,f}^{O_{aft}}$ stores only $s.st$ (and $s.id$).

Relationship STARTS According to Allen's algebra, a STARTS selection query reports all intervals that *start* where q does, i.e., with $q.st = s.st$, but outlive its *end*, i.e., with $q.end < s.end$. By construction, HINT^m stores such intervals as originals in the first relevant partition. We consider two cases for every index level ℓ. If $f = l$, we report each interval in the $P_{\ell,f}^{O_{in}}$ subdivision that satisfies both query conditions and each interval in $P_{\ell,f}^{O_{aft}}$ that satisfies only $q.st = s.st$; for the latter intervals, their $s.end$ is by construction after $q.end$. So, we report $\{s \in P_{\ell,f}^{O_{in}}: q.st = s.st \wedge q.end < s.end\} \cup \{s \in P_{\ell,f}^{O_{aft}}: q.st = s.st\}$. In contrast, if $f \neq l$, the results can only come from the intervals that end after the first relevant partition at current level ℓ, i.e., from $P_{\ell,f}^{O_{aft}}$. But, as subdivisions $P_{\ell,f}^{O_{aft}}$ store only $s.st$ according to Table 3, we cannot directly check the $q.end < s.end$ condition. Instead, we rely on the replicas inside the last relevant partition at any index level. Intuitively, if an interval $\{s \in P_{\ell,f}^{O_{aft}}: q.st = s.st\}$ is stored as a replica in the last relevant partition l at a level ℓ', which either (1) ends inside l (i.e., $s \in P_{\ell',l}^{R_{in}}$) but after $q.end$ or (2) outlives the partition (i.e., $s \in P_{\ell',l}^{R_{aft}}$) then

---

[9] In the general case, $\ell \neq \ell'$ holds for levels ℓ and ℓ'.

$q.end < s.end$ holds for s. The above two sets are computed as $\bigcup_{\forall \ell'} \left\{ \left\{ s \in P_{\ell',l}^{R_{in}} : q.end < s.end \right\} \bigcup P_{\ell',l}^{R_{aft}} \right\}$.

Relationship `STARTED_BY`. As an inverse to `STARTS`, a `STARTED_BY` selection determines all intervals that again *start* at $q.st$ but *end* before $q.end$. Therefore, if $f = l$ holds at a level ℓ, we consider only the intervals that *both* start and end inside the partition, reporting set $\left\{ s \in P_{\ell,f}^{O_{in}} : q.st = s.st \wedge q.end > s.end \right\}$. Otherwise, results are found among all originals in f. For the $P_{\ell,f}^{O_{in}}$ subdivision, we directly output $\left\{ s \in P_{\ell,f}^{O_{in}} : q.st = s.st \right\}$ as their $s.end$ is by construction before $q.end$. For the intervals in $s \in P_{\ell,f}^{O_{aft}}$ with $q.st = s.st$, we apply a similar technique to `STARTS` for checking the $q.end > s.end$ condition. Intuitively, such an interval s will be reported if it ends at any level ℓ', either inside a partition i with $f < i < l$ or in the last relevant partition l but before $q.end$. For this purpose, we check if s is inside set $\bigcup_{\forall \ell'} \left\{ \left\{ \bigcup_{\forall f < i < l} P_{\ell',i}^{R_{in}} \right\} \bigcup \left\{ s \in P_{\ell',l}^{R_{in}} : q.end > s.end \right\} \right\}$.

Relationship `FINISHES` This selection query returns all intervals that *end* exactly where query q does, i.e., with $q.end = s.end$, but *start* before q, i.e., with $q.st > s.st$. If q overlaps a single partition ($f = l$) at a level ℓ, we consider the intervals that end in the last relevant partition l: $\left\{ s \in P_{\ell,l}^{O_{in}} : q.end = s.end \wedge q.st > s.st \right\} \bigcup \left\{ s \in P_{\ell,l}^{R_{in}} : q.end = s.end \right\}$. Otherwise ($f \neq l$), only replicas that end inside partition l (Subdivision $P_{\ell,l}^{R_{in}}$) with $q.end = s.end$ can be part of the results. To this end, we face a similar challenge to `STARTS`/`STARTED_BY` as $P_{\ell,l}^{R_{in}}$ does not store $s.st$ (see Table 3) to directly check $q.st > s.st$. The solution is to check if an interval $\left\{ s \in P_{\ell,l}^{R_{in}} : q.end = s.end \right\}$ is contained in set $\bigcup_{\forall \ell'} \left\{ \left\{ s \in P_{\ell',f}^{O_{aft}} : q.st > s.st \right\} \bigcup P_{\ell',f}^{R_{aft}} \right\}$, i.e., the intervals that either (1) start before $q.st$ in the first relevant partition f at any level ℓ' or (2) are stored in $P_{\ell',f}^{R_{aft}}$ and so, their start is by construction before $q.st$.

Relationship `FINISHED_BY`. A `FINISHED_BY` selection inverses the second condition of `FINISHES`, determining intervals with $q.end = s.end$ and $q.st < s.st$. For a level ℓ, if $f = l$ we report the intervals that start and end inside the partition, and satisfy both conditions, i.e., set $\left\{ s \in P_{\ell,l}^{O_{in}} : q.end = s.end \wedge q.st < s.st \right\}$. Otherwise ($f \neq l$), the results are among all intervals that end in partition l, i.e., set $\left\{ s \in P_{\ell,l}^{O_{in}} : q.end = s.end \right\} \bigcup \left\{ s \in P_{\ell,l}^{R_{in}} : q.end = s.end \right\}$. For the intervals from subdivision $P_{\ell,l}^{O_{in}}$, $q.st < s.st$ holds by construction while for $P_{\ell,l}^{R_{in}}$ intervals, a direct check of the condition is not

possible. Instead, we check such an interval s against the set of intervals that start either (1) after q in the first relevant partition at any level ℓ' or (2) inside the partitions in between the first and the last relevant; set $\bigcup_{\forall \ell'} \left\{ \left\{ s \in P_{\ell',f}^{O_{aft}} : q.st < s.st \right\} \bigcup \left\{ \bigcup_{\forall f < i < l} P_{\ell',i}^{O_{aft}} \right\} \right\}$.

Relationship `MEETS` This selection query returns all intervals that *start* at $q.end$. Under this, we report for each level ℓ, all originals in the last relevant partition l that satisfy the $q.end = s.st$ condition, i.e., set $\left\{ s \in P_{\ell,l}^{O_{in}} \bigcup P_{\ell,l}^{O_{aft}} : q.end = s.st \right\}$.

Relationship `MET_BY` This selection query returns all intervals that *end* at $q.st$. To this end, the results are among the intervals that end inside the first relevant partition f, i.e., set $\left\{ s \in P_{\ell,f}^{O_{in}} \bigcup P_{\ell,f}^{R_{in}} : q.st = s.end \right\}$, at each level ℓ.

Relationship `OVERLAPS` An `OVERLAPS` selection determines all non-disjoint intervals to query q, which start *after* $q.st$ and end *after* $q.end$. If q overlaps a single partition ($f = l$) at a level ℓ, such intervals are found among the originals in the partition; for the $P_{\ell,f}^{O_{in}}$ subdivision all query conditions are checked, while for an s in $P_{\ell,f}^{O_{aft}}$, $q.end < s.end$ always holds. So, we report set $\left\{ s \in P_{\ell,f}^{O_{in}} : q.st < s.st \wedge q.end > s.st \wedge q.end < s.end \right\} \bigcup \left\{ s \in P_{\ell,f}^{O_{aft}} : q.st < s.st \wedge q.end > s.st \right\}$. Otherwise, results are reported in two parts. The first part is drawn from the originals in the last relevant partition at each level ℓ, i.e., $\left\{ s \in P_{\ell,l}^{O_{in}} : q.end > s.st \wedge q.end < s.end \right\} \bigcup \left\{ s \in P_{\ell,l}^{O_{aft}} : q.end > s.st \right\}$. For the second part, we consider the intervals that start before partition l and outlive q, i.e., set $\left\{ \left\{ s \in P_{\ell,l}^{R_{in}} : q.end < s.end \right\} \bigcup P_{\ell,l}^{R_{aft}} \right\}$. For every such interval s, $q.end > s.st$ holds by construction, but we need to check its start against $q.st$. As subdivisions $P_{\ell}^{R_{in}}$ and $P_{\ell}^{R_{aft}}$ do not store $s.st$, we cannot directly check the $q.st < s.st$ condition. Instead, we compare s against all $P_{\ell'}^{O_{aft}}$ at any level ℓ' that (1) either start before $q.st$ in the first relevant partition f or (2) inside every partition in between f and l, i.e., set $\bigcup_{\forall \ell'} \left\{ \left\{ s \in P_{\ell',f}^{O_{aft}} : q.st < s.st \right\} \bigcup \left\{ \bigcup_{\forall f < i < l} P_{\ell',i}^{O_{aft}} \right\} \right\}$.

Relationship `OVERLAPPED_BY` As inverse to `OVERLAPS`, the `OVERLAPPED_BY` selection determines all non-disjoint intervals to q that start *before* $q.st$ and end *before* $q.end$. If $f = l$, we draw the results from all intervals (both originals and replicas) that *end* inside the partition; set $\left\{ s \in P_{\ell,f}^{O_{in}} : q.st > s.st \wedge q.st < s.end \wedge q.end > s.end \right\} \bigcup \left\{ s \in P_{\ell,f}^{R_{in}} : q.st < s.end \wedge q.end > s.end \right\}$. Otherwise, the results consist of two parts for every level ℓ. The first part includes again originals and replicas that end inside the first relevant partition f, but now, condition

$q.end > s.end$ always holds by construction. Hence, we report set $\left\{s \in P_{\ell,f}^{O_{in}} : q.st > s.st \wedge q.st < s.end\right\} \bigcup \left\{s \in P_{\ell,f}^{R_{in}} : q.st < s.end\right\}$. For the second part, we seek results among all intervals that start before q, i.e., originals $\left\{s \in P_{\ell,f}^{O_{aft}} : q.st > s.st\right\}$ and replicas $P_{\ell,f}^{R_{aft}}$ for both sets $q.st < s.end$ holds by construction as intervals outlive the first relevant partition f. As neither of the $P_{\ell,f}^{O_{aft}}$ and $P_{\ell,f}^{R_{aft}}$ subdivisions maintains $s.end$, we check $q.end > s.end$ by determining the replicas at any index level ℓ' that end (1) either before the last relevant partition l or (2) inside l after $q.end$, i.e., set $\bigcup_{\forall \ell'} \left\{\left\{\bigcup_{\forall f < i < l} P_{\ell',i}^{R_{in}}\right\} \bigcup \left\{s \in P_{\ell',l}^{R_{in}} : q.end > s.end\right\}\right\}$.

Relationship CONTAINS This selection query returns all intervals, fully *contained* inside the query interval q, i.e., with $q.st < s.st \wedge q.end > s.end$. For every level ℓ, if $f = l$, q can contain only intervals that *both* start and end in this partition, i.e., from subdivision $P_{\ell,f}^{O_{in}}$; we report set $\left\{s \in P_{\ell,f}^{O_{in}} : q.st < s.st \wedge q.end > s.end\right\}$. Otherwise, the results are drawn from the original intervals in every partition from the first relevant partition f to the last l; for the latter only originals that end inside the partition are considered. Specifically, for the intervals in $P_{\ell}^{O_{in}}$ subdivisions, we report $\left\{s \in P_{\ell,f}^{O_{in}} : q.st < s.st\right\} \bigcup \left\{\bigcup_{\forall f < i < l} P_{\ell,i}^{O_{in}}\right\} \bigcup \left\{s \in P_{\ell,l}^{O_{in}} : q.end > s.end\right\}$; observe how only one condition is checked for partitions f and l, while for every partition i in between, all originals that end inside i are directly output. In contrast, for all intervals in the $P_{\ell}^{O_{aft}}$ subdivisions, we need to check the $q.end > s.end$ condition; additionally, for every $s \in P_{\ell,f}^{O_{aft}}$ subdivision, we also check if $q.st < s.st$ holds. As $P_{\ell}^{O_{aft}}$ subdivisions store only $s.st$, $q.end < s.end$ is checked similarly to OVERLAPPED_BY, i.e., using set $\bigcup_{\forall \ell'} \left\{\left\{\bigcup_{\forall f < i < l} P_{\ell',i}^{R_{in}}\right\} \bigcup \left\{s \in P_{\ell',l}^{R_{in}} : q.end > s.end\right\}\right\}$.

Relationship CONTAINED_BY This selection determines all intervals that fully *contain* q, i.e., with $q.st > s.st \wedge q.end < s.end$. For each level ℓ, if $f = l$, the result intervals are found among all subdivisions in the partition, reporting $\left\{s \in P_{\ell,f}^{O_{in}} : q.st > s.st \wedge q.end < s.end\right\} \bigcup \left\{s \in P_{\ell,f}^{O_{aft}} : q.st > s.st\right\} \bigcup \left\{s \in P_{\ell,f}^{R_{in}} : q.end < s.end\right\} \bigcup P_{\ell,f}^{R_{aft}}$. In contrast, if $f \neq l$, the results are among the intervals that (1) start before $q.st$, corresponding to set $\left\{s \in P_{\ell,f}^{O_{aft}} : q.st > s.st\right\} \bigcup P_{\ell,f}^{R_{aft}}$, and (2) end after $q.end$. As the $P_{\ell}^{O_{aft}}$ or the $P_{\ell}^{R_{aft}}$ subdivisions do not store $s.end$, in order to check the $q.end < s.end$ condition, we need to inter-

that either end inside the last relevant partition l or outlive it, i.e., set $\bigcup_{\forall \ell'} \left\{\left\{s \in P_{\ell',l}^{R_{in}} : q.end < s.end\right\} \bigcup P_{\ell',l}^{R_{aft}}\right\}$.

Relationship BEFORE A BEFORE selection determines all intervals that start *after* q. Such intervals are found at each level ℓ as originals either (1) inside the last relevant partition l, if they satisfy $q.end < s.st$, i.e., set $\left\{s \in P_{\ell,l}^{O_{in}} \bigcup P_{\ell,l}^{O_{aft}} : q.end < s.st\right\}$ or (2) inside every partition after l, i.e., set $\bigcup_{\forall i > l} \left\{P_{\ell,i}^{O_{in}} \bigcup P_{\ell,i}^{O_{aft}}\right\}$. Note that replicas from these partitions are ignored as they will only produce duplicate results.

Relationship AFTER An AFTER selection determines all intervals that end *before* q. Results are found at each level among the intervals which end inside either (1) the first relevant partition f and satisfy $q.st > s.end$, i.e., set $\left\{s \in P_{\ell,f}^{O_{in}} \bigcup P_{\ell,f}^{R_{in}} : q.st > s.end\right\}$ or (2) every partition before f, i.e., set $\bigcup_{\forall i < f} \left\{P_{\ell,i}^{O_{in}} \bigcup P_{\ell,i}^{R_{in}}\right\}$. Note that subdivisions $P_{\ell,i}^{O_{aft}}$ and $P_{\ell,i}^{R_{aft}}$ are ignored to avoid duplicate results (Table 12).

6.2 One setup for all

The storage optimization discussed in Sect. 4.1.2 allows the G-OVERLAPS setup of HINT^m to reduce the memory footprint of the index and improve cache locality. But as an optimization technique tailored for the G-OVERLAPS relationship, it has a negative impact on Allen's algebra basic relationships. The key issue is that we cannot directly check the conditions on $s.end$ for the $P^{O_{aft}}$ and $P^{R_{aft}}$ subdivisions and on $s.st$ for $P^{R_{in}}$ and $P^{R_{aft}}$. Instead, we are forced to access extra partitions to implicitly conduct these checks, e.g., the $P_{\ell',l}^{R_{in}}$ and $P_{\ell',l}^{R_{aft}}$ subdivisions in the last relevant partition l at each index level ℓ', for the STARTS relationship.

In view of this, we next consider a *subs+sort* setup of HINT^m for Allen's algebra.[10] Essentially, no changes are required if query q overlaps a single partition ($f = l$) at a level ℓ as all necessary information is available for the selection conditions. Further, the computation of MEETS, MET_BY, BEFORE and AFTER queries remains unchanged. So, in what follows, we discuss the necessary changes for the rest of relationships in the $f \neq l$ case.

Relationship EQUALS We can now directly retrieve results from the first relevant partition f and the $P_{\ell,f}^{O_{aft}}$ subdivision by checking both query conditions, i.e., we report set $\left\{s \in P_{\ell,f}^{O_{aft}} : q.st = s.st \wedge q.end = s.end\right\}$.

---

[10] The cache misses and the skewness & sparsity optimizations are orthogonal and can be straightforwardly combined with the subs+sort HINT^m setup.

Table 12 Allen's algebra relationships, 'One setup for all'

q REL s	Definition	Result set
EQUALS	$q.st = s.st \wedge$ $q.end = s.end$	**if** $f = l,$ $\left\{ s \in P_{\ell,f}^{O_{in}} : q.st = s.st \wedge q.end = s.end \right\}$ **else** $\left\{ s \in P_{\ell,f}^{O_{aft}} : q.st = s.st \wedge q.end = s.end \right\}$
STARTS	$q.st = s.st \wedge$ $q.end < s.end$	$\forall \ell:$ **if** $f = l,$ $\left\{ s \in P_{\ell,f}^{O_{in}} : q.st = s.st \wedge q.end < s.end \right\} \bigcup \left\{ s \in P_{\ell,f}^{O_{aft}} : q.st = s.st \right\}$ **else** $\left\{ s \in P_{\ell,f}^{O_{aft}} : q.st = s.st \wedge q.end < s.end \right\}$
STARTED_BY	$q.st = s.st \wedge$ $q.end > s.end$	$\forall \ell:$ **if** $f = l,$ $\left\{ s \in P_{\ell,f}^{O_{in}} : q.st = s.st \wedge q.end > s.end \right\}$ **else** $\left\{ s \in P_{\ell,f}^{O_{aft}} : q.st = s.st \right\} \bigcup \left\{ s \in P_{\ell,f}^{O_{aft}} : q.st = s.st \wedge q.end > s.end \right\}$
FINISHES	$q.end = s.end \wedge$ $q.st > s.st$	$\forall \ell:$ **if** $f = l,$ $\left\{ s \in P_{\ell,f}^{O_{in}} : q.end = s.end \wedge q.st > s.st \right\} \bigcup \left\{ s \in P_{\ell,l}^{R_{in}} : q.end = s.end \right\}$ **else** $\left\{ s \in P_{\ell,l}^{R_{in}} : q.end = s.end \wedge q.st > s.st \right\}$
FINISHED_BY	$q.end = s.end \wedge$ $q.st < s.st$	$\forall \ell:$ **if** $f = l,$ $\left\{ s \in P_{\ell,l}^{O_{in}} : q.end = s.end \wedge q.st < s.st \right\} \bigcup \left\{ s \in P_{\ell,l}^{R_{in}} : q.end = s.end \wedge q.st < s.st \right\}$ **else** $\left\{ s \in P_{\ell,l}^{O_{in}} : q.end = s.end \wedge q.st < s.st \right\}$
MEETS	$q.end = s.st$	$\forall \ell:$ $\left\{ s \in P_{\ell,l}^{O_{in}} \bigcup P_{\ell,l}^{O_{aft}} : q.end = s.st \right\}$
MET_BY	$q.st = s.end$	$\forall \ell:$ $\left\{ s \in P_{\ell,f}^{O_{in}} \bigcup P_{\ell,f}^{R_{in}} : q.st = s.end \right\}$
OVERLAPS	$q.st < s.st \wedge$ $q.end > s.st \wedge$ $q.end < s.end$	$\forall \ell:$ **if** $f = l,$ $\left\{ s \in P_{\ell,f}^{O_{in}} : q.st < s.st \wedge q.end > s.st \right\}$ $\left\{ s \in P_{\ell,f}^{O_{aft}} : q.st < s.st \wedge q.end > s.st \right\}$ **else** $\left\{ s \in P_{\ell,l}^{O_{in}} : q.end > s.st \wedge q.st < s.st \wedge q.end < s.end \right\} \bigcup \left\{ s \in P_{\ell,l}^{O_{aft}} : q.end > s.st \right\} \bigcup$ $\left\{ s \in P_{\ell,l}^{R_{in}} : q.st < s.st \wedge q.end < s.end \right\} \bigcup \left\{ s \in P_{\ell,l}^{R_{aft}} : q.st < s.st \right\}$
OVERLAPPED_BY	$q.st > s.st \wedge$ $q.st < s.end$ $q.end > s.end$	$\forall \ell:$ **if** $f = l,$ $\left\{ s \in P_{\ell,f}^{O_{in}} : q.st > s.st \wedge q.end > s.end \right\}$ $\left\{ s \in P_{\ell,f}^{R_{in}} : q.st < s.end \wedge q.end > s.end \right\}$ **else** $\left\{ s \in P_{\ell,f}^{O_{in}} : q.st > s.st \wedge q.st < s.end \wedge q.end > s.end \right\} \bigcup \left\{ s \in P_{\ell,f}^{R_{in}} : q.st < s.end \right\} \bigcup$ $\left\{ s \in P_{\ell,f}^{O_{aft}} : q.st > s.st \wedge q.end > s.end \right\} \bigcup \left\{ s \in P_{\ell,f}^{R_{aft}} : q.end > s.end \right\}$

Relationship STARTS With $s.end$ in $P_{\ell,f}^{O_{aft}}$, both query conditions can be directly checked at each level ℓ and thus report $\left\{s \in P_{\ell,f}^{O_{aft}} : q.st = s.st \wedge q.end < s.end\right\}$.

Relationship STARTED_BY Similar to STARTS, we can directly check both conditions for $P_{\ell,f}^{O_{aft}}$ in the first relevant partition f. We report $\left\{s \in P_{\ell,f}^{O_{in}} : q.st = s.st\right\} \bigcup \left\{s \in P_{\ell,f}^{O_{aft}} : q.st = s.st \wedge q.end > s.end\right\}$, at each level.

Relationship FINISHES With $s.st$ in $P_{\ell,l}^{R_{in}}$ subdivisions, we can directly check $q.st > s.st$ and report $\left\{s \in P_{\ell,l}^{R_{in}} : q.end = s.end \wedge q.st > s.st\right\}$, at each level.

Relationship FINISHED_BY Similar to FINISHES, we can directly check both conditions on $P_{\ell,l}^{R_{in}}$ and thus, report at each level ℓ, set $\left\{s \in P_{\ell,l}^{O_{in}} : q.end = s.end\right\} \bigcup \left\{s \in P_{\ell,l}^{R_{in}} : q.end = s.end \wedge q.st < s.st\right\}$.

Relationship OVERLAPS With $s.st$ in subdivisions $P_{\ell}^{R_{in}}$ and $P_{\ell}^{R_{aft}}$, we directly check $q.st < s.st$ for partition l. So, we report $\left\{s \in P_{\ell,l}^{R_{in}} : q.st < s.st \wedge q.end < s.end\right\} \bigcup \left\{s \in P_{\ell,l}^{R_{aft}} : q.st < s.st\right\}$ intervals at each level along with the set $\left\{s \in P_{\ell,l}^{O_{in}} : q.end > s.st \wedge q.end < s.end\right\} \bigcup \left\{s \in P_{\ell,l}^{O_{aft}} : q.end > s.st\right\}$.

Relationship OVERLAPPED_BY With $s.end$ stored in $P_{\ell,f}^{O_{aft}}$ and $P_{\ell,f}^{R_{aft}}$, we can directly check $q.end > s.end$, reporting set $\left\{s \in P_{\ell,f}^{O_{aft}} : q.st > s.st \wedge q.end > s.end\right\} \bigcup \left\{s \in P_{\ell,f}^{R_{aft}} : q.end > s.end\right\}$ along with the intervals contained in $\left\{s \in P_{\ell,f}^{O_{in}} : q.st > s.st \wedge q.st < s.end\right\} \bigcup \left\{s \in P_{\ell,f}^{R_{in}} : q.st < s.end\right\}$.

Relationship CONTAINS With $s.end$ in $P_{\ell}^{O_{aft}}$ subdivisions, we can directly check the $q.end > s.end$ condition to output $\left\{s \in P_{\ell,f}^{O_{aft}} : q.st < s.st \wedge q.end > s.end\right\} \bigcup \left\{s \in \bigcup_{\forall f < i < l} P_{\ell,i}^{O_{aft}} : q.end > s.end\right\}$ along with the set $\left\{s \in P_{\ell,f}^{O_{in}} : q.st < s.st\right\} \bigcup \left\{s \in P_{\ell,l}^{O_{in}} : q.end > s.end\right\} \bigcup \left\{\bigcup_{\forall f < i < l} P_{\ell,i}^{O_{in}}\right\}$ from $P_{\ell}^{O_{in}}$ subdivisions at each level.

Relationship CONTAINED_BY With $s.end$ stored in both $P_{\ell,f}^{O_{aft}}$ and $P_{\ell,f}^{R_{aft}}$ subdivisions, we can now directly check the $q.end < s.end$ condition at each level ℓ, reporting the intervals $\left\{s \in P_{\ell,f}^{O_{aft}} : q.st > s.st \wedge q.end < s.end\right\} \bigcup \left\{s \in P_{\ell,f}^{R_{aft}} : q.end < s.end\right\}$.

Table 12 continued

q REL s	Definition	Result set
CONTAINS	$q.st < s.st \wedge$ $q.end > s.end$	$\forall \ell:$ **if** $f = l,$ $\left\{s \in P_{\ell,f}^{O_{in}} : q.st < s.st \wedge q.end > s.end\right\}$ **else** $\left\{s \in P_{\ell,f}^{O_{in}} : q.st < s.st\right\} \bigcup \left\{s \in P_{\ell,f}^{O_{aft}} : q.st < s.st \wedge q.end > s.end\right\} \bigcup$ $\left\{\bigcup_{f < i < l} P_{\ell,i}^{O_{in}}\right\} \bigcup \left\{s \in \bigcup_{f < i < l} P_{\ell,i}^{O_{aft}} : q.end > s.end\right\} \bigcup$ $\left\{s \in P_{\ell,l}^{O_{in}} : q.end > s.end\right\}$
CONTAINED_BY	$q.st > s.st \wedge$ $q.end < s.end$	$\forall \ell:$ **if** $f = l,$ $\left\{s \in P_{\ell,f}^{O_{in}} : q.st > s.st \wedge q.end < s.end\right\} \bigcup \left\{s \in P_{\ell,f}^{O_{aft}} : q.st > s.st\right\} \bigcup P_{\ell,f}^{R_{aft}}$ $\left\{s \in P_{\ell,f}^{R_{in}} : q.end < s.end\right\}$ **else** $\left\{s \in P_{\ell,f}^{O_{aft}} : q.st > s.st \wedge q.end < s.end\right\} \bigcup P_{\ell,f}^{R_{aft}}$
BEFORE	$q.end < s.st$	$\forall \ell: \left\{s \in P_{\ell,l}^{O_{in}} \cup P_{\ell,l}^{O_{aft}} : q.end < s.st\right\} \bigcup \left\{\bigcup_{\forall i > l} \left\{P_{\ell,i}^{O_{in}} \cup P_{\ell,i}^{O_{aft}}\right\}\right\}$
AFTER	$q.st > s.end$	$\forall \ell: \left\{s \in P_{\ell,f}^{O_{in}} \cup P_{\ell,f}^{R_{in}} : q.st > s.end\right\} \bigcup \left\{\bigcup_{\forall i < f} \left\{P_{\ell,i}^{O_{in}} \cup P_{\ell,i}^{R_{in}}\right\}\right\}$

6.3 Bottom-up evaluation approach

Both setups of HINT^m can benefit from the bottom-up approach in Sect. 3.2.2. The idea is to determine the levels when the last bit of the first (last) relevant partition f (l) are set to 1 or 0, for the first time. Due to lack of space, we discuss only STARTS for the G-OVERLAPS setup as an example. Specifically, results are found among the original intervals stored in the first relevant partition f up to the level where the last bit in f is 1, for the first time. All originals in f at a higher level start by construction of the index before $q.st$ and thus, violate $q.st = s.st$. Further, at levels after the one where the last bit of l is 0 for the first time, $q.end < s.end$ always holds for all $s \in P_{\ell',l}^{R_{in}}$. Consider the query q in Fig. 7. Candidate results are contained only as originals in $P_{4,5}$, where the last bit of $f = 5$ is 1. Also as the last bit of l is 0 at the 4th level, all $P^{R_{in}}$ intervals in $P_{2,2}$, $P_{1,1}$, $P_{0,0}$ satisfy $q.end < s.end$.

7 Experiments on Allen's algebra

For the second part of our experiments, we focus on selection queries under the basic relationships of Allen's algebra. We first compare the two alternative HINT^m setups from Sect. 6 and then put the best setup against the competition. We extended our code for all competitive indices in Sect. 5 to support Allen's algebra. We ran our tests on datasets BOOKS, WEBKIT, TAXIS and GREEND; we omit the results on the synthetic datasets due to lack of space and because of observing similar trends. Lastly, parameter m and all other index parameters are set according to Table 7.

7.1 Determining the best index setup

Figure 17 reports the throughputs achieved by the two HINT^m setups; results in WEBKIT and GREEND are similar and therefore omitted due to lack of space. Note that both setups adopt the bottom-up evaluation (Sect. 3.2.2) and employ the *skewness & sparsity* and the *cache misses* optimizations (Sects. 4.2 and 4.3). The 'one setup for all' setup drastically improves the performance of HINT^m for the majority of the queries. Essentially, the G-OVERLAPS setup matches the performance of 'one setup for all' in the G-OVERLAPS relationship, as expected, and in relationships where only one partition per level is examined by both setups, without the need to indirectly check a condition, i.e., in MEETS, MET_BY, BEFORE and AFTER. In the rest, 'one setup for all' is from one to several orders of magnitude faster.

We also compare the two setups on their index size and updates. As expected, 'one setup for all' requires more space (35% on average, see Table 13) due to disabling *sopt* from Sect. 4.1.2. Regarding insertions and deletions, both setups
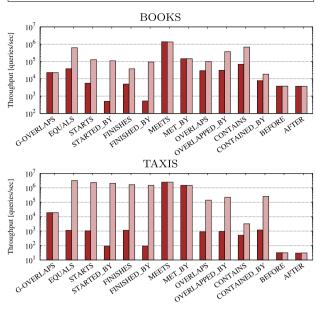

Fig. 17 Comparing HINT^m setups: throughput

Table 13 Comparing HINT^m setups: index size (MBs)

HINT^m setup	BOOKS	WEBKIT	TAXIS	GREEND
For G-OVERLAPS	81	98	2039	1278
One setup for all	134	138	2769	1321

will employ the hybrid setting in Sect. 4.4 with similar performance. Overall, in a typical space-time tradeoff, 'one setup for all' increases the space requirements in exchange of drastically accelerating querying, even several orders of magnitude for some relationships. So, for the rest of our analysis, HINT^m operates always under 'one setup for all'.

7.2 Index performance comparison

Figure 18 compares the performance of all studied indices. The first 4 rows of plots report the results for OVERLAPS, OVERLAPPED_BY, CONTAINS, CONTAINED_BY, while varying the query extent, similar to Fig. 14. Note that for CONTAINED_BY on TAXIS and GREEND, we consider a different range of values as these datasets contain significantly shorter intervals than BOOKS and WEBKIT. The last row of plots reports on the rest of the relationships where the selection queries essentially resemble typical stabbing queries, i.e., query overlaps either one partition per level or only two in total partitions in EQUALS.

Overall, HINT^m exhibits the highest throughput for all queries based on Allen's algebra relationships, in line with the results in Fig. 14. Its performance gap to the competitor indices ranges from almost half to several orders of magni-
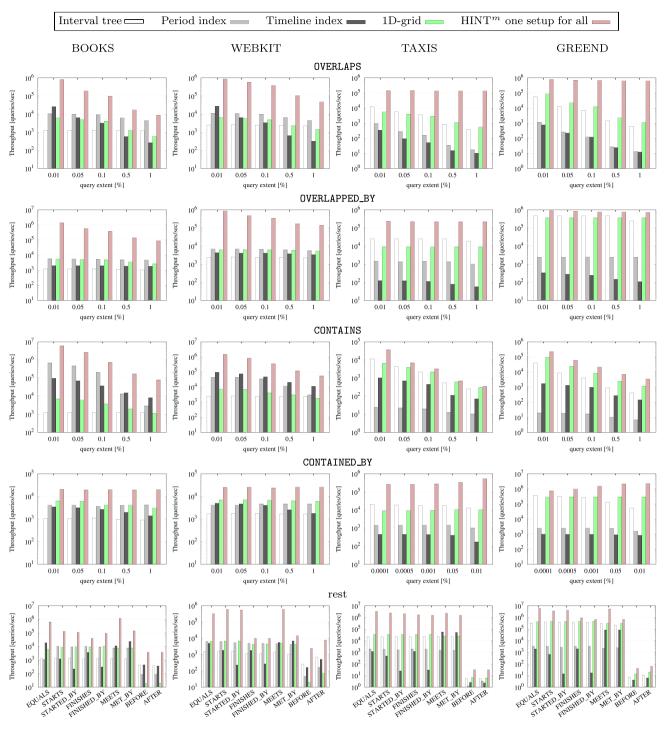
Fig. 18 Comparing throughputs on Allen's algebra, real datasets

tude. Essentially, the smallest performance gap is observed mainly in WEBKIT and GREEND where the input intervals are very short.

8 Conclusions

We proposed a hierarchical index (HINT) for intervals, which has low space complexity and minimizes the number of data accesses and comparisons during query evaluation. Our experiments on real and synthetic datasets show that HINT outperforms previous work by almost one order of magni-

tude in a wide variety of interval data and query distributions. Our index fully supports selection queries based on Allen's relationships [1] between intervals, achieving consistently excellent performance independently of the query predicate. In the future, we intend to extend our work toward multiple directions. Regarding the index structure, we plan to consider compression techniques to further reduce HINT^m memory footprint and adaptive variants that e.g., use non-regular partitioning to better deal with long intervals. We also plan to support queries combining temporal selections and selections on additional attributes or the interval duration [4]. Further, we will study how to manage transactional data using HINT, e.g., streams of events. Last, we plan to investigate hardware-aware techniques, e.g., for effective parallelization relying on the fact that HINT partitions are independent, or near-storage computation with Processing-in-Memory.

References

1. Allen, J.F.: Maintaining knowledge about temporal intervals. Commun. ACM **26**(11), 832–843 (1983)
2. Awad, A., Tommasini, R., Langhi, S., Kamel, M., Valle, E.D., Sakr, S.: D^{2}IA: user-defined interval analytics on distributed streams. Inf. Sys. **104**, 101679 (2022)
3. Becker, B., Gschwind, S., Ohler, T., Seeger, B., Widmayer, P.: An asymptotically optimal multiversion b-tree. VLDB J. **5**(4), 264–275 (1996)
4. Behrend, A., Dignös, A., Gamper, J., Schmiegelt, P., Voigt, H., Rottmann, M., Kahl, K.: Period index: a learned 2d hash index for range and duration queries. In: SSTD, pp. 100–109 (2019)
5. de Berg, M., Cheong, O., van Kreveld, M.J., Overmars, M.H.: Computational geometry: algorithms and applications, 3rd edn. Springer, Berlin (2008)
6. Böhlen, M.H., Dignös, A., Gamper, J., Jensen, C.S.: Temporal data management - an overview. In: eBISS, pp. 51–83 (2017)
7. Bouros, P., Lampropoulos, K., Tsitsigkos, D., Mamoulis, N., Terrovitis, M.: Band joins for interval data. In: EDBT, pp. 443–446 (2020)
8. Bouros, P., Mamoulis, N.: A forward scan based plane sweep algorithm for parallel interval joins. Proc. VLDB Endow. **10**(11), 1346–1357 (2017)
9. Bouros, P., Mamoulis, N.: Interval count semi-joins. In: EDBT, pp. 425–428 (2018)
10. Bouros, P., Mamoulis, N., Tsitsigkos, D., Terrovitis, M.: In-memory interval joins. VLDB J. **30**(4), 667–691 (2021)
11. Cafagna, F., Böhlen, M.H.: Disjoint interval partitioning. VLDB J. **26**(3), 447–466 (2017)
12. Chekol, M.W., Pirrò, G., Stuckenschmidt, H.: Fast interval joins for temporal SPARQL queries. In: ACM WWW, pp. 1148–1154 (2019)
13. Christodoulou, G., Bouros, P., Mamoulis, N.: HINT: a hierarchical index for intervals in main memory. In: ACM SIGMOD, pp. 1257–1270 (2022)
14. Dalvi, N.N., Suciu, D.: Efficient query evaluation on probabilistic databases. In: VLDB, pp. 864–875 (2004)
15. Dignös, A., Böhlen, M.H., Gamper, J.: Overlap interval partition join. In: ACM SIGMOD, pp. 1459–1470 (2014)
16. Dignös, A., Böhlen, M.H., Gamper, J., Jensen, C.S., Moser, P.: Leveraging range joins for the computation of overlap joins. VLDB J. **31**(1), 75–99 (2022)
17. Dittrich, J., Seeger, B.: Data redundancy and duplicate detection in spatial join processing. In: IEEE ICDE, pp. 535–546 (2000)
18. Edelsbrunner, H.: Dynamic rectangle intersection searching. Tech. Rep. 47, Institute for Information Processing, Technical University of Graz, Austria (1980)
19. Ferragina, P., Vinciguerra, G.: The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. Proc. VLDB Endow. **13**(8), 1162–1175 (2020)
20. Höppner, F., Peter, S.: Temporal interval pattern languages to characterize time flow. Int. Rev. Data Min. Knowl. Disc. **4**(3), 196–212 (2014)
21. Kaufmann, M., Manjili, A.A., Vagenas, P., Fischer, P.M., Kossmann, D., Färber, F., May, N.: Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In: ACM SIGMOD, pp. 1173–1184 (2013)
22. Kline, N., Snodgrass, R.T.: Computing temporal aggregates. In: IEEE ICDE, pp. 222–231 (1995)
23. Körber, M., Glombiewski, N., Morgen, A., Seeger, B.: TPStream: low-latency and high-throughput temporal pattern matching on event streams. Distrib. Parallel Datab. **39**(2), 361–412 (2021)
24. Kriegel, H., Pötke, M., Seidl, T.: Managing intervals efficiently in object-relational databases. In: VLDB, pp. 407–418 (2000)
25. Lomet, D.B.: Scheme for invalidating references to freed storage. IBM J. Res. Dev. **19**(1), 26–35 (1975)
26. Lomet, D.B., Hong, M., Nehme, R.V., Zhang, R.: Transaction time indexing with version compression. Proc. VLDB Endow. **1**(1), 870–881 (2008)
27. Min, J., Park, M., Chung, C.: XPRESS: a queriable compression for XML data. In: ACM SIGMOD, pp. 122–133 (2003)
28. Monacchi, A., Egarter, D., Elmenreich, W., D'Alessandro, S., Tonello, A.M.: GREEND: an energy consumption dataset of households in italy and austria. In: SmartGridComm, pp. 511–516 (2014)
29. Moon, B., López, I.F.V., Immanuel, V.: Efficient algorithms for large-scale temporal aggregation. IEEE TKDE **15**(3), 744–759 (2003)
30. Overmars, M.H.: The Design of Dynamic Data Structures. Lecture Notes in Computer Science, vol. 156. Springer, Berlin (1983)

31. Pagel, B., Six, H., Toben, H., Widmayer, P.: Towards an analysis of range query performance in spatial data structures. In: ACM PODS, pp. 214–221 (1993)
32. Piatov, D., Helmer, S.: Sweeping-based temporal aggregation. In: SSTD, pp. 125–144 (2017)
33. Piatov, D., Helmer, S., Dignös, A.: An interval join optimized for modern hardware. In: IEEE ICDE, pp. 1098–1109 (2016)
34. Piatov, D., Helmer, S., Dignös, A., Persia, F.: Cache-efficient sweeping-based interval joins for extended allen relation predicates. VLDB J. **30**(3), 379–402 (2021)
35. Salzberg, B., Tsotras, V.J.: Comparison of access methods for time-evolving data. ACM Comp. Surv. **31**(2), 158–221 (1999)
36. Samarati, P., Sweeney, L.: Generalizing data to provide anonymity when disclosing information (abstract). In: ACM PODS, p. 188 (1998)
37. Snodgrass, R.T., Ahn, I.: Temporal databases. Computer **19**(9), 35–42 (1986)
38. Zhu, K., Fletcher, G.H.L., Yakovets, N., Papapetrou, O., Wu, Y.: Scalable temporal clique enumeration. In: SSTD, pp. 120–129 (2019)