Querying Interval Data on Steroids

Panagiotis Bouros, George Christodoulou, Christian Rauch, Artur Titkov, Nikos Mamoulis

Abstract—A wide range of applications manage interval data with selections and overlap joins being the two most fundamental querying operations. Selection queries are typically evaluated using interval indexing. However, the state-the-of-art HINT index and its competitors, are only designed to service single query requests while modern systems receive a large number of queries at the same time. In view of this challenge, we study the batch processing of selection queries on HINT. We propose two novel strategies termed level-based and partition-based, which both operate in a per-level fashion, i.e., they collect the results for all queries at an index level before moving to the next. The new strategies reduce the cache misses when climbing the index hierarchy, and in particular, partition-based can prevent scanning every index partition more than once. Our experiments on real-world intervals showed that our batch strategies always outperform a baseline which executes queries in a serial fashion, and that partition-based is overall the most efficient one. Motivated by our shared computation techniques for query batches, we also study overlap joins anew across the entire spectrum of different setups, based on the (pre)-existence of interval indexing. For unindexed inputs, we enhance the state-of-the-art optFS join algorithm with effective partitioning proposed for HINT and for indexed inputs, we propose a novel algorithm HINT-join which concurrently scans the HINT input indices, joining partition pairs with optFS. Our tests showed the advantage of HINT-join over indexed nested-loops solutions that employ either B$^+$-trees or probing a single HINT even if powered by our partition-based batch processing.

Index Terms—Interval data, query processing, range queries, selections, overlap joins, batch processing

1 Introduction

Given a discrete or continuous 1D space, an interval is defined by a starting and an ending point in this domain. For instance, in the space of all non-negative integers $\mathbb{N}$, an interval $[st, end]$ with st, $end \in \mathbb{N}$ and $st \leq end$, is the subset of $\mathbb{N}$, which includes all integers x with $st \leq x \leq end$.[1] Two intervals $r = [r.st, r.end]$ and $s = [s.st, s.end]$ *overlap*, denoted by $r \cap s \neq \emptyset$, if they share at least one common point, i.e., if $s.st \leq t.st \leq s.end$ or $t.st \leq s.st \leq q.end$. Collections of intervals (associated with objects) are found in a wide range of applications; e.g., in temporal databases [1], [2], where each tuple has a *validity interval* to capture the time period when the tuple is valid. In statistics and probabilistic databases [3], *uncertain* values are often approximated by (confidence or uncertainty) intervals. In data anonymization [4], attribute values are often generalized to value ranges. 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) can be modeled as intervals [6], while input streams can be joined over relative time intervals [7].

The two most fundamental querying operations on intervals are range (selection) queries and overlap joins; Figure 1 exemplifies these operations for a temporal database

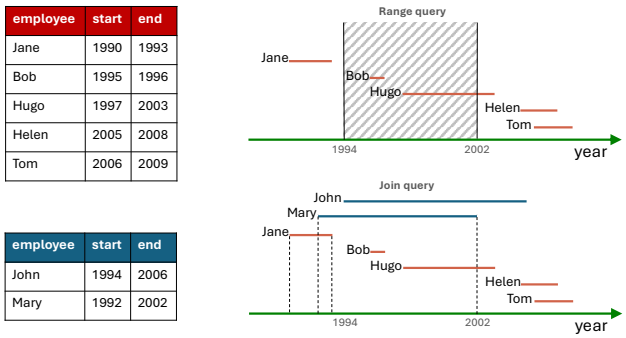

Fig. 1: Querying intervals

which stores the employees of a company employed at two departments (red and blue) during different time periods. Consider the red department; a range query would retrieve all employees whose employment periods overlap the time window from 1994 to 2002, i.e., Bob and Hugo. Formally, given a collection of intervals $\mathcal{S}$ and a query interval $q = [q.st, q.end]$, a selection query $\sigma_{[q.st,q.end]}(\mathcal{S})$ retrieves all $s \in \mathcal{S}$ intervals with $s \cap q \neq \emptyset$. Now consider both departments in the figure; an interval join would determine all pairs of employees, whose employment periods overlap, e.g., Mary from the blue department and Jane from the red. Formally, the $\mathcal{R} \bowtie \mathcal{S}$ join retrieves all (r, s) pairs of intervals from $\mathcal{R} \times \mathcal{S}$, such that $r \cap s \neq \emptyset$.

Motivation. A plethora of methods are proposed for efficiently computing selection and join queries on intervals. Data structures, such as the interval tree [8], an 1D-grid, the period index [9], the RD-index [10], and HINT [11] divide the domain into disjoint partitions and assign the intervals to them. Upon querying, only the partitions that overlap the query range are examined to report results. Despite their efficiency, these indexing structures are designed to service single-query requests, i.e., execute queries in a serial fashion by independently scanning overlapping partitions. How-

- *P. Bouros, C. Rauch and A. Titkov are with the Institute of Computer Science, Johannes Gutenberg University Mainz, Germany. E-mail: {bouros, c.rauch, artitkov}@uni-mainz.de*
- *G. Christodoulou is with Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Netherlands. E-mail: g.c.christodoulou@tudelft.nl*
- *N. Mamoulis is with the Department of Electrical & Computer Engineering, University of Ioannina, Greece. E-mail: nikos@cse.uoi.gr*

1. Note that the intervals in this paper are closed. Yet, our techniques and discussions apply on generic intervals where the *start* and *end* sides are either open or closed.

ever, modern transactional databases, OLTP systems and cloud services (e.g., by Amazon and Google) must deal with query-heavy workflows receiving thousands or millions of queries per second.[2] A straightforward serial evaluation of queries under such a setting (1) is cache agnostic and results in cache misses and (2) has no potential of accessing data partitions once for multiple queries. Instead, modern systems opt for processing the queries in batches to save resources and reduce the overall time. AWS for instance allows users to run batch jobs on their analytical services, e.g., Amazon RedShift.[3] *Multi* or *batch* query processing relies on specialized computation sharing techniques to reduce the total execution time of a batch and minimize cache misses. Such techniques are widely used e.g., for traditional relational data [12], [13], [14], spatial data [15], [16], [17] and graphs [18], [19], [20], and in IR systems [21], [22].

For interval joins, previous solutions either follow a nested-loops approach [23], [24], employ partitioning [25], [26], [27], [28] or interval indexing [29], [30], [31], [32], or build upon the plane-sweep approach from computational geometry [33], [34], [35], [36]. Nevertheless, we observe three issues with previous work on joins. First, a large part of it, e.g., [24], [23], [25], [27], [31] target disk-resident data and so, their goal is to minimize I/O accesses during the join. Such a setting is less relevant in contemporary in-memory data management. Second, although plane-sweep based evaluation was shown in [33] [34], [35] to outperform the alternatives, it was never paired with efficient interval indexing. Last, recent developments in partitioning and interval indexing proposed in [11], [37] have never been applied to boost the computation of interval joins.

Contributions. We present a framework for efficiently computing massive workflows of selection queries. We build on the state-of-the-art interval index HINT [11], [37], which is shown to be typically an order of magnitude faster than the competition, minimizing the number of data accesses and comparisons. Moreover, HINT also exhibits low space complexity, while offering a competitive building time.

We focus on batch processing of selection queries; to the best of our knowledge, this is the first work that investigates such a setting for interval data. We propose two novel evaluation strategies, termed *level-based* and *partition-based*, which capitalize on HINT's structure. The strategies operate on a per-level fashion, i.e., they first evaluate all queries for an index level before moving to the next, which achieves better cache locality. We also enable access sharing for partition-based, which allows us to scan the contents of every index partition only once. Our tests showed that our batch strategies achieve on average, a 30% performance improvement for inputs with long intervals and a 50% with short ones, over the serial execution of the query-based baseline. When access sharing is also activated on top of the partition-based strategy, the batch processing becomes an order of magnitude faster than the serial execution.

Batch query processing also resembles a join, where the query batch is set as the second join input. Although such an approach allows for sharing computations among objects, it is competitive only when the batch size is smaller than the cardinality of the input collection [37]. Under this premise, we study overlap joins anew across the entire spectrum of different setups, based on the (pre)-existence of interval indexing; we consider again the state-of-the-art HINT. For unindexed inputs, we enhanced the state-of-the-art algorithm optFS [35] to adopt the effective partitioning techniques employed also for the partitions in HINT; our tests showed a 25% performance improvement on average by this enhanced optFS, with the exception of joining extremely short intervals where the join is in fact cheap to begin with. When inputs are indexed by HINT, we propose a novel join algorithm termed HINT-join, which concurrently scans the input index hierarchies and joins pairs of partitions as unindexed inputs using optFS. Our tests showed that HINT-join achieves an average performance enhancement of 25% for inputs with long intervals and 50% with short ones, compared to indexed nested-loops solutions that use HINT indexing powered by our partition-based batch processing strategy with access sharing or B$^+$-trees [32].

TABLE 1: Notation summary

notation	description
s (r)	interval
$s.id$, $s.st$, $s.end$	identifier, start and end point of interval s
$\mathcal{S}$ ($\mathcal{R}$)	collection of intervals
$prefix(k, x)$	k-bit prefix of integer x
$P_{\ell,i}$	i-th partition at level ℓ of HINT
$P_{\ell,f}$ ($P_{\ell,l}$)	first (last) relevant partition at level ℓ
$P^O_{\ell,i}$ ($P^R_{\ell,i}$)	division of $P_{\ell,i}$ with originals (replicas)
$P^{O_{in}}_{\ell,i}$ ($P^{O_{aft}}_{\ell,i}$)	intervals in $P^O_{\ell,i}$ ending inside (after) $P_{\ell,i}$
$P^{R_{in}}_{\ell,i}$ ($P^{R_{aft}}_{\ell,i}$)	intervals in $P^R_{\ell,i}$ ending inside (after) $P_{\ell,i}$
$\sigma_{[q.st,q.end]}(\mathcal{S})$	selection query
$\mathcal{R} \bowtie \mathcal{S}$	join query

Comparison to our previous publication. This article significantly extends our preliminary work in [38]. First, we devise a novel variant of the partition-based strategy for batch processing selection queries on HINT, which guarantees every index partition is accessed only once for the entire query batch. Second, we study the efficient evaluation of interval overlap joins. For unindexed inputs, we extend the state-of-the-art optFS to use the subdivisions proposed in [11] and the domain partitioning scheme, previously considered only for the parallel computation. Last, we propose the HINT-join algorithm when both inputs are indexed by HINT.

Outline. Section 2 provides the necessary background on interval indexing and joins. Then, Section 3 details our strategies for batch processing selection queries while Section 4 presents our solutions for overlap joins, distinguishing between the case of indexed or unindexed inputs. Section 5 presents our experimental analysis. Finally, Section 6 discusses related work and Section 7 concludes our study.

2 Background

We revisit the state-of-the-art for indexing and joining intervals. Table 1 summarizes the notation used in the text.

2.1 Indexing Intervals with HINT

HINT [11], is a hierarchical index for intervals, utilizing their binary representation. Parameter m indicates the number of

2. https://aws.amazon.com/blogs/aws/amazon-s3-two-trillion-objects-11-million-requests-second/

3. https://docs.aws.amazon.com/wellarchitected/latest/analytics-lens/ batch-data-processing.html

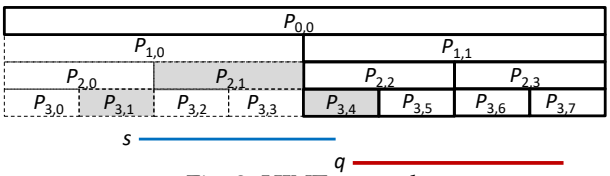

Fig. 2: HINT example

bits for representing intervals, which results in establishing $m+1$ levels. Figure 2 exemplifies HINT for $m=3$ and a hierarchy of 4 index levels. Each level ℓ ($0 \leq \ell \leq m$), uniformly divides the domain into 2^ℓ partitions. As we ascend the HINT hierarchy, each level ℓ corresponds to a more significant bit of the binary representation. Hence, the number of partitions in each level decreases by a factor of 2 while covering double the size. During the insertion process, every interval s undergoes normalization and discretization within the $[0, 2^m-1]$ domain and is inserted to at most 2 partitions per level. If a given interval spans more than 2 partitions at a specific level, it is assigned to an upper level, where partitions cover a larger part of the domain. Overall, the assignment principle is based on selecting the *smallest set* of partitions across all levels that collectively cover an interval s. Moreover, intervals in each partition P are split into two divisions: those that start *inside* P (called *originals*), denoted by P^O, and those that start *before* P (called *replicas*), denoted by P^R. For instance, in Figure 2, interval s is added to the shaded partitions; in $P_{3,1}$, s is added to the $P^O_{3,1}$ division which stores original intervals in $P_{3,1}$, while in $P_{2,1}$ and $P_{3,4}$, interval s is stored to the corresponding replica divisions, i.e., $P^R_{2,1}$ and $P^R_{3,4}$, respectively.

Given a selection query $q=\sigma_{[q.st,q.end]}(\mathcal{S})$, at each index level ℓ only the sequence of partitions $P_{\ell,i}$ that intersect q are accessed; we call these, the *relevant* partitions of q. For query q in Figure 2, only the partitions with a solid/bold outline will be accessed. To avoid duplicate results, originals and replicas divisions are only accessed for the first relevant partition at each level ℓ, while for the remaining partitions only originals are considered. Finally, the endpoints of an interval s are compared to query q only for the first and the last relevant partition at a level; for every (original) interval s inside the rest, intermediate partitions $q.st < s.st < q.end$ holds, by construction of the index.

Bottom-up traversal. We further reduce the number of partitions where comparisons are required by traversing HINT in a *bottom-up* fashion, instead of a conventional *top-down*. Under the bottom-up traversal, the expected number of partitions requiring comparisons is 4, according to [11]. Consider again Figure 2. For query q, no comparisons are needed in partition $P_{2,3}$, because all intervals assigned to $P_{2,3}$ should overlap with $P_{3,6}$ and the extent of $P_{3,6}$ is covered by q. Hence, the start of all intervals in $P_{2,3}$ is guaranteed to be before $q.end$ (which is inside $P_{3,7}$).

Algorithm 1 illustrates how HINT evaluates a selection query, in a bottom-up fashion. The algorithm uses two auxiliary flags, $compfirst$ and $complast$ to mark if it is necessary to perform comparisons at the current level (and all levels above it), for the first and the last relevant partition, respectively, At each level ℓ, the sequence of relevant partitions to the query q is determined in Lines 4–5, based on the ℓ-prefixes of $q.st$ and $q.end$, denoted by f and l, respectively. Each relevant partition $P_{\ell,i}$ (and its divisions) is then processed against q by the $Process$ function in Lines 12–26. For the first relevant partition $P_{\ell,f}$ both originals $P^O_{\ell,f}$ and replicas $P^R_{\ell,f}$ are accessed. If $f=l$, i.e., the first and the last relevant partitions coincide, and both $compfirst$, $complast$ are set, then comparisons are needed for both $P^O_{\ell,f}$ and $P^R_{\ell,f}$. Otherwise, if $complast$ is only set, the algorithm safely skips the $q.st \leq s.end$ comparisons, while if $compfist$ is only set, regardless whether $f=l$, we only perform $q.st \leq s.end$ comparisons to both $P^O_{\ell,f}$ and $P^R_{\ell,f}$. If neither flag is set, then all intervals in the first relevant partition are simply reported as results. When the last partition $P_{\ell,l}$ is examined and $l > f$ (Lines 23–24) the algorithm considers $P^O_{\ell,l}$ and checks only if $s.st \leq q.end$ for each interval there. Lastly, for every partition in-between the first and the last one, all original intervals are simply reported.

ALGORITHM 1: Selection query on HINT

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

1 $compfirst \leftarrow TRUE$;
2 $complast \leftarrow TRUE$;
3 **foreach** *level* $\ell = m$ **to** 0 **do** ▷ bottom-up fashion
4 $f \leftarrow prefix(\ell, q.st)$; ▷ first overlapping partition
5 $l \leftarrow prefix(\ell, q.end)$; ▷ last overlapping partition
6 **foreach** *partition* $i = f$ *to* l **do**
7 $ProcessPartition(\mathcal{H}.P_{\ell,i}, q, compfirst, complast, f, l)$;
8 **if** f *mod* $2 = 0$ **then** ▷ last bit of f is 0
9 $compfirst \leftarrow FALSE$;
10 **if** l *mod* $2 = 1$ **then** ▷ last bit of l is 1
11 $complast \leftarrow FALSE$;

Function $Process$(*partition* $P_{\ell,i}$, *query* q, *flag* $compfirst$, *flag* $complast$, f, l):
12 **if** $i = f$ **then**
13 **if** $i = l$ **and** $compfirst$ **and** $complast$ **then**
14 **output** $\{s \in P^O_{\ell,i} : q.st \leq s.end \wedge s.st \leq q.end\}$;
15 **output** $\{s \in P^R_{\ell,i} : q.st \leq s.end\}$;
16 **else if** $i = l$ **and** $complast$ **then**
17 **output** $\{s \in P^O_{\ell,i} : s.st \leq q.end\}$;
18 **output** $\{s \in P^R\}$;
19 **else if** $compfirst$ **then**
20 **output** $\{s \in P^O_{\ell,i} \bigcup P^R_{\ell,i} : q.st \leq s.end\}$;
21 **else**
22 **output** $\{s \in P^O_{\ell,i} \bigcup P^R_{\ell,i}\}$;
23 **else if** $i = l$ **and** $complast$ **then** ▷ $l > f$
24 **output** $\{s \in P^O_{\ell,i} : s.st \leq q.end\}$;
25 **else** ▷ in-between or last ($l > f$), no comparisons
26 **output** $\{s \in P^O_{\ell,i}\}$;

Optimizations. A series of optimizations were proposed in [11] to boost the query processing on HINT. First, the number of performed comparisons are reduced by further dividing the P^O and P^R divisions of a partition P. Specifically, P^O is split into *subdivisions* $P^{O_{in}}$ and $P^{O_{aft}}$, so that $P^{O_{in}}$ ($P^{O_{aft}}$) holds the intervals from P^O that *end* inside (resp. after) P. Similarly, each P^R is divided into $P^{R_{in}}$ and $P^{R_{aft}}$. Second, the *storage* optimization reduces the memory footprint of the index. So far, each interval s is stored as a $\langle s.id, s.st, s.end\rangle$ triplet. But, only the $P^{O_{in}}$ subdivisions require both endpoints. For $P^{O_{aft}}$ and $P^{R_{in}}$, $s.st$ and $s.end$ are only needed, respectively, while for $P^{R_{aft}}$, none of the endpoints are required, as no comparisons are performed. Another optimization to save on comparisons is to keep

ALGORITHM 2: Forward Scan Plane Sweep (FS)

```
Input  : collections of intervals R and S
Output : set of all overlapping interval pairs (r, s) ∈ R × S
1  sort R and S by st endpoint;                          ▷ for plane-sweep
2  r ← first interval in R;
3  s ← first interval in S;
4  while R and S not depleted do
5      if r.st < s.st then
6          s' ← s;
7          while s' ≠ null and r.end ≥ s'.st do
8              output (r, s');                           ▷ update result
9              s' ← next interval in S;                  ▷ scan forward
10         r ← next interval in R;
11     else
12         r' ← r;
13         while r' ≠ null and s.end ≥ r'.st do
14             output (r', s);                           ▷ update result
15             r' ← next interval in R;                  ▷ scan forward
16         s ← next interval in S;
```

the subdivisions sorted; each using its own *beneficial sorting*. Due to *data skewness & sparsity*, many partitions may be empty, especially at the lowest levels. To deal with this, HINT merges the contents of all P^O divisions at the same level ℓ into a single table T^O_ℓ and builds an auxiliary index which is used to access non-empty divisions upon querying. The last optimization deals with potential *cache misses* while traversing the index. As no comparisons are needed at most of the levels, HINT stores the id and the *endpoints* of an interval separately. When no comparisons are needed, the index directly reports results from the id array.

2.2 Joining Intervals with optFS

The state-of-the-art optFS [34], [35] for overlap interval joins builds upon the plane-sweep from [39][4], which performs forward scans directly on the input collections. Algorithm 2 illustrates the pseudo-code of this plane-sweep implementation, denoted by FS. Initially, the input collections $\mathcal{R}$, $\mathcal{S}$ are sorted by the starting endpoint of the intervals. Then, FS sweeps a line, which stops at each st point in both inputs. For each position of the sweep line, i.e., the start of an interval, say $r \in \mathcal{R}$, the FS produces join results by pairing r with all intervals from the opposite collection, that start (1) after the sweep line and (2) before $r.end$, i.e., all $s' \in \mathcal{S}$ with $r.st \leq s'.st \leq r.end$ (internal while-loops in Lines 7–9 and 13–15). To boost the performance of FS, the authors in [34], [35] proposed the following four optimizations.

Grouping. The key idea is to group consecutively sweeped intervals from the same input and produce join results in batches, avoiding redundant comparisons. Assume that the next interval to consider is $r \in \mathcal{R}$, i.e., $r.st < s.st$ (the other case is symmetric). Starting from r, FS accesses all $r' \in \mathcal{R}$ with $r.st < s.st$ to form a group $G_\mathcal{R}$; the contents of $G_\mathcal{R}$ are reordered by increasing end endpoint. Then, FS performs a forward scan on $\mathcal{S}$, but for the entire group $G_\mathcal{R}$, instead of interval r. Every interval from $\mathcal{S}$ that overlaps an interval $r_i \in G_\mathcal{R}$, is guaranteed to also overlap all intervals after r_i in $G_\mathcal{R}$, without the need for further comparisons.

Bucket indexing. For this optimization, the domain is first split into a number of equally-sized disjoint stripes; all intervals from $\mathcal{R}$ (resp. $\mathcal{S}$) that start inside a particular stripe are stored in a dedicated bucket of the $\mathcal{BI}_\mathcal{R}$ (resp. $\mathcal{BI}_\mathcal{S}$) index. With bucket indexing, the forward scan of FS now examines the contents of buckets. Consider current interval $r \in \mathcal{R}$. FS finds the bucket $B \in \mathcal{BI}_\mathcal{S}$ which covers $r.end$ and then, scans each $s' \in \mathcal{S}$ inside all buckets before B to directly produce results, without any comparisons. In contrast, the contents of bucket B are examined similar to Algorithm 2.

4. Originally for 2D rectangle joins but reduced for 1D intervals.

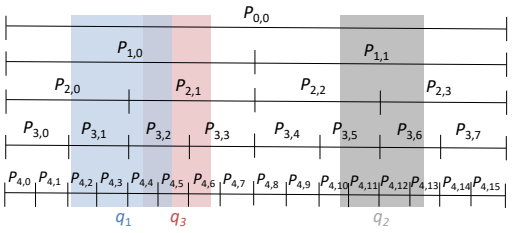

Fig. 3: Running example

Enhanced loop unrolling. The third optimization builds upon the loop unrolling/unwinding code technique [40], [41], [42], which reduces the execution time by (1) eliminating the overhead of controlling a loop (i.e., checking its exit condition) and the latency due to main memory accesses, and (2) reducing branch penalties. To enhance FS, the forward scans are unrolled by a 32 factor and the join conditions are checked only once every 32 intervals.

Decomposed layout. The last technique is inspired by the Decomposition Storage Model (DSM) [43], in column-oriented database systems [44]. The idea is to modify how the intervals are stored to reduce the footprint of FS and the number of cache misses. Instead of storing an input as an array of $\langle id, st, end \rangle$ tuples, we maintain two separate st and end arrays. With this decomposition, FS iterates only over the st arrays when advancing the sweep line or forward scanning, and over the end arrays for the interval groups.

optFS: a self-tuning FS. The above optimizations can be combined but their effect depends on the selectivity of the join. optFS is a self-tuning variant of FS, that decides online which of the optimizations should be activated. For this purpose, optFS relies on sampling to measure the average extent of the conducted forward scans, as an indicator for the join selectivity. When forward scans cover only some tens (or a hundred in the worst case) of intervals on average then grouping, bucket indexing and the decomposed data layout will not payoff and therefore, optFS deactivates them. In contrast, enhanced loop unrolling is always activated.

3 Batch Processing Selection Queries

Given a collection of intervals $\mathcal{S}$ indexed by HINT, and a batch of selection queries $\mathcal{Q}$, we next discuss four evaluation strategies. Without loss of generality and for illustration purposes, we describe the strategies using a HINT without the optimizations from Section 2.1. To exemplify the strategies, we use the index and the $\mathcal{Q} = \{q_1, q_2, q_3\}$ batch in Figure 3. For each query q, we highlight its relevant (i.e., overlapping) partitions on each level based on its $[q.st, q.end]$ range.

3.1 Query-based

A straightforward approach for processing $\mathcal{Q}$ is to independently compute each query in a serial fashion, using

TABLE 2: Access patterns for the queries in Figure 3; for each query, relevant partitions are accordingly colored; partitions colored in black are accessed only once for all relevant queries

Strategy	Accessed partitions
Query-based	$P_{4,2} \to P_{4,3} \to P_{4,4} \to P_{4,5} \to P_{3,1} \to P_{3,2} \to P_{2,0} \to P_{2,1} \to P_{1,0} \to P_{0,0} \to$ $P_{4,10} \to P_{4,11} \to P_{4,12} \to P_{4,13} \to P_{3,5} \to P_{3,6} \to P_{2,2} \to P_{2,3} \to P_{1,1} \to P_{0,0} \to$ $P_{4,4} \to P_{4,5} \to P_{4,6} \to P_{3,2} \to P_{3,3} \to P_{2,1} \to P_{1,0} \to P_{0,0}$
Query-based with sorting	$P_{4,2} \to P_{4,3} \to P_{4,4} \to P_{4,5} \to P_{3,1} \to P_{3,2} \to P_{2,0} \to P_{2,1} \to P_{1,0} \to P_{0,0} \to$ $P_{4,4} \to P_{4,5} \to P_{4,6} \to P_{3,2} \to P_{3,3} \to P_{2,1} \to P_{1,0} \to P_{0,0} \to$ $P_{4,10} \to P_{4,11} \to P_{4,12} \to P_{4,13} \to P_{3,5} \to P_{3,6} \to P_{2,2} \to P_{2,3} \to P_{1,1} \to P_{0,0}$
Level-based with sorting	$P_{4,2} \to P_{4,3} \to P_{4,4} \to P_{4,5} \to P_{4,4} \to P_{4,5} \to P_{4,6} \to P_{4,10} \to P_{4,11} \to P_{4,12} \to P_{4,13} \to$ $P_{3,1} \to P_{3,2} \to P_{3,2} \to P_{3,3} \to P_{3,5} \to P_{3,6} \to$ $P_{2,0} \to P_{2,1} \to P_{2,1} \to P_{2,2} \to P_{2,3} \to$ $P_{1,0} \to P_{1,0} \to P_{1,1} \to$ $P_{0,0} \to P_{0,0} \to P_{0,0}$
Partition-based with sorting	$P_{4,2} \to P_{4,3} \to P_{4,4} \to P_{4,4} \to P_{4,5} \to P_{4,5} \to P_{4,6} \to P_{4,10} \to P_{4,11} \to P_{4,12} \to P_{4,13} \to$ $P_{3,1} \to P_{3,2} \to P_{3,2} \to P_{3,3} \to P_{3,5} \to P_{3,6} \to$ $P_{2,0} \to P_{2,1} \to P_{2,1} \to P_{2,2} \to P_{2,3} \to$ $P_{1,0} \to P_{1,0} \to P_{1,1} \to$ $P_{0,0} \to P_{0,0} \to P_{0,0}$
Partition-based with sorting & access sharing	$P_{4,2} \to P_{4,3} \to \mathbf{P_{4,4}} \to \mathbf{P_{4,5}} \to P_{4,6} \to P_{4,10} \to P_{4,11} \to P_{4,12} \to P_{4,13} \to$ $P_{3,1} \to \mathbf{P_{3,2}} \to P_{3,3} \to P_{3,5} \to P_{3,6} \to$ $P_{2,0} \to \mathbf{P_{2,1}} \to P_{2,2} \to P_{2,3} \to$ $\mathbf{P_{1,0}} \to P_{1,1} \to$ $P_{0,0}$

Algorithm 1. We call this strategy, *query-based*. Despite its simplicity, a major shortcoming of query-based is that the strategy operates in a cache agnostic fashion. As every query q typically overlaps multiple partitions from different levels of the index, the computation of all queries in $\mathcal{Q}$ requires accessing data in different parts of the memory. Hence, the memory access pattern is prone to cache misses.

Consider our running example in Figure 3; the query-based strategy will execute the queries in the order of their subscript. The first row in Table 2 illustrates the occurred access pattern, i.e., the order in which the partitions of the index will be accessed. First, all relevant partitions for q_1 are accessed on each level of the index (highlighted in blue), following the bottom-up approach from [11]; similarly, the relevant partitions for q_2 (highlighted in gray) are accessed next. The two sets of partitions are located on opposite sides of the index, which causes several "jumps" to different parts of the memory; we refer to these jumps as *horizontal*. Finally, for query q_3, we need to "jump" back to the front part of the index to access the partitions highlighted in red.

To improve the above access pattern, one solution is to execute the queries according to their starting point $q.st$. In Figure 3, the query-based strategy will now execute first q_1, followed by q_3 and last, q_2. The modified access pattern is depicted in the second row of Table 2. This new pattern enables us to finish with all queries accessing partitions in the front part of the index, before moving to the back.

3.2 Level-based

Sorting the queries by their start will reduce cache misses caused by horizontal jumps and therefore, will enhance the query-based strategy. However, the bottom-up approach employed for each query will still incur cache misses because of the *vertical* jumps in the index. For example in case of the adjacent q_1 and q_3 queries in Figure 3, we have to first climb all levels of the index to compute q_1 and then, start over from the bottom level for q_3.

To deal with these vertical jumps, we propose a different strategy which capitalizes on the fact that partitions in HINT are physically organized in a level-based fashion. The *level-based* strategy still builds upon the bottom-up approach but the evaluation process proceeds to the next level of the index only after the relevant partitions for all queries in batch $\mathcal{Q}$ are already accessed and processed to report potential results. Algorithm 3 shows the pseudocode of this strategy, essentially extending Algorithm 1 in two ways. First, we maintain a $compfirst[q]$ and a $complast[q]$ flag for each query q in batch $\mathcal{Q}$, which are initialized in Lines 2–3 and updated in Lines 10–13 at each level, according to the last bits of the first relevant partition f and the last l. Second, we introduce in Line 5, a new `for-loop` to iterate over all queries in the batch, at current level ℓ. Each query q is then processed for all relevant partitions at ℓ in Lines 8–9 using the $Process$ function from Algorithm 1 Similarly to query-based, the level-based strategy can also benefit from sorting the queries by their start, avoiding the horizontal jumps when accessing the relevant partitions at each level. Returning to our running example, the third row in Table 2 depicts the access pattern for the level-based strategy, with sorting activated. To better illustrate the effect of the strategy, we write the sequence of accessed partitions in five lines, one for each level of the index. Notice how on every line (index level), the evaluation switches from the relevant partitions of q_1, to the ones of q_3 and finally, to q_2, before moving to the next level. Under this premise, we avoid the vertical jumps incurred by independently applying the bottom-up approach in the query-based strategy.

ALGORITHM 3: Level-based strategy

Input : HINT index $\mathcal{H}$, batch of queries $\mathcal{Q}$
Output : set of all overlapping intervals, for each $q \in \mathcal{Q}$

1 **foreach** *query* $q \in \mathcal{Q}$ **do** ▷ Initialization
2 $compfirst[q] \leftarrow TRUE$;
3 $complast[q] \leftarrow TRUE$;
4 **foreach** *level* $\ell = m$ **to** 0 **do** ▷ bottom-up fashion
5 **foreach** *query* $q \in \mathcal{Q}$ **do**
6 $f \leftarrow prefix(\ell, q.st)$; ▷ first overlapping partition
7 $l \leftarrow prefix(\ell, q.end)$; ▷ last overlapping partition
8 **foreach** *partition* $i = f$ *to* l **do**
9 $Process(\mathcal{H}.P_{\ell,i}, q, compfirst[q], complast[q], f, l)$;
▷ From Algorithm 1
10 **if** f *mod* $2 = 0$ **then** ▷ last bit of f is 0
11 $compfirst[q] \leftarrow FALSE$;
12 **if** l *mod* $2 = 1$ **then** ▷ last bit of l is 1
13 $complast[q] \leftarrow FALSE$;

3.3 Partition-based

Despite evaluating queries on a per-level basis and examining the queries by their start, jumps can still occur in the level-based strategy. Consider again the access pattern of level-based in Table 2; specifically, the first line which corresponds to the bottom level of the index. The strategy will access partitions $P_{4,4}$ and $P_{4,5}$ first for q_1 and then again for q_2, in the exact same order. To deal with this type of horizontal jumps, we next introduce the *partition-based* strategy. Similar to level-based, the partition-based strategy adopts the per-level evaluation and can benefit from sorting the queries, but it processes independently every partition. Intuitively, in order to proceed to the next partition on a level, all queries relevant to the current partition must be first evaluated. Algorithm 4 illustrates the pseudocode of the strategy. As the key difference to Algorithm 3, the partition-based strategy introduces a new `for-loop` to iterate over all partitions on current level ℓ, in Line 5. Notice

ALGORITHM 4: Partition-based strategy

```
   Input  : HINT index H, batch of queries Q
   Output : set of all overlapping intervals, for each q ∈ Q
1  foreach query q ∈ Q do                                   ▷ Initialization
2      compfirst[q] ← TRUE;
3      complast[q] ← TRUE;
4  foreach level ℓ = m to 0 do                              ▷ bottom-up fashion
5      foreach partition i on level ℓ do
6          foreach relevant query q ∈ Q to partition i do
7              f ← prefix(ℓ, q.st);                         ▷ first overlapping
                partition
8              l ← prefix(ℓ, q.end);                        ▷ last overlapping
                partition
9              Process(H.P_{ℓ,i}, q, compfirst[q], complast[q], f, l);
                   ▷ From Algorithm 1
24     foreach q ∈ Q do
25         if f mod 2 = 0 then                              ▷ last bit of f is 0
26             compfirst[q] ← FALSE;
27         if l mod 2 = 1 then                              ▷ last bit of l is 1
28             complast[q] ← FALSE;
```

how Algorithm 3 iterates over each query in batch $\mathcal{Q}$ for current level ℓ, while Algorithm 4 iterates over all relevant queries in batch $\mathcal{Q}$ (Line 6) for current partition i, i.e., all queries whose range overlaps with i, on the current level. These relevant queries are executed in Line 9 using again $Process$ from Algorihtm 1, for current partition i.

The fourth row in Table 2 shows the access pattern for the partition-based strategy. If we compare this pattern to the level-based, we observe that when processing the bottom level of the index, the partition-based strategy will first finish with partition $P_{4,4}$ for both queries q_1 and q_3, then access $P_{4,5}$, for the same queries and finally, move on to partition $P_{4,10}$. Note that despite applying a partition-based evaluation at each level, the contents of $P_{4,7}$, $P_{4,8}$, $P_{4,9}$ and $P_{3,4}$ will be never scanned as no query overlaps with them.

Last, we elaborate on Line 6 of Algorithm 4 and the fast computation of the relevant queries for current partition i. A simple approach would compare each query in $\mathcal{Q}$ to partition i, incurring extra costs. Instead, we rely on the cheap bitwise operations that determine the first and the last relevant partitions of a query. We define for each partition i, a range of relevant queries; for this, the queries need to be examined in increasing order of their start. The range of i's relevant queries starts from the first query q with $prefix(\ell, q.st) = i$ to the last with $prefix(\ell, q.end) = i$.

3.4 Access Sharing

Partition-based is able to achieve good cache locality by reducing the number of horizontal and vertical jumps that occur on the index. The strategy however is unable to prevent multiple accesses of the same index entries.[5] Consider once again our running example in Figure 3 and the fourth row in Table 2. The intervals of all partitions relevant to both q_1 and q_3 (e.g., $P_{4,4}$ and $P_{4,5}$ in the fourth level) will be accessed two times; the partition-based strategy can only guarantee that these two access operations will take place one right after the other to reduce cache misses. To prevent accessing HINT entries multiple times, we next propose an extension to the partition-based strategy which allows access sharing among the queries in the query batch.

Algorithm 5 provides a high-level pseudocode of the partition-based variant with access sharing. Despite the differences in the code structure compared to Algorithm 4, the key principle of processing every partition before moving to the next on the level remains the same, i.e., the `for-loops` in Lines 1 and 2. For every partition $P_{\ell,i}$ in a level ℓ, we distinguish among three types of relevant queries; essentially, we breakdown the relevant queries determined in Line 6 of Algorithm 4. We denote by $\mathcal{Q}_f$ and $\mathcal{Q}_l$, the queries in the query batch $\mathcal{Q}$ for which $P_{\ell,i}$ acts as the first and the last relevant partition, respectively.[6] We also denote by $\mathcal{Q}_c$ the rest of the relevant queries, i.e., those that contain current partition $P_{\ell,i}$. Then, to share accesses while processing $P_{\ell,i}$, it suffices to compute a series of joins between the original and replica divisions of $P_{\ell,i}$ and the $\mathcal{Q}_f$, $\mathcal{Q}_l$, $\mathcal{Q}_c$ sets. Specifically, following the principles for evaluating selection queries in HINT discussed in Section 2.1, the $P^O_{\ell,i}$ originals are joined with all three query sets, while replicas $P^R_{\ell,i}$, only with $\mathcal{Q}_l$. The remaining combinations are dropped to avoid duplicate results. The last row in Table 2 details the access pattern of the partition-based strategy with access sharing. We mark in black font the partitions for which accessing is shared. Observe for instance, how sharing allows the strategy to access $P_{4,4}$ and $P_{4,5}$ only once, in order to produce results for both q_1 and q_3 at the same time.

For the joins in Lines 6–7, we can directly employ the optFS algorithm from [34], [35] (see Section 2.2), provided that the queries in batch $\mathcal{Q}$ (and thus, also in $\mathcal{Q}_f$ and $\mathcal{Q}_l$) are sorted by their start. Note that $P^O_{\ell,i}$ is already sorted by *start* because of the beneficial sorting optimization discussed in Section 2.1.The plane-sweep approach of optFS guarantees that every interval in $P^O_{\ell,i}$ is accessed and compared against the queries, only once. On the other hand, the join in Line 8 is essentially a cross product as all original intervals in $P_{\ell,i}$ overlap with every query that contains the partition, by definition. This product can be efficiently computed by appending the contents of $P^O_{\ell,i}$ to the result set of each query in $\mathcal{Q}_f$. Lastly, for the join in Line 9, we cannot directly employ optFS as $P^R_{\ell,i}$ and $\mathcal{Q}_f$ use different sorting; specifically, the intervals in $P^R_{\ell,i}$ are sorted by their *end* [11], [37] while $\mathcal{Q}_f$ is sorted by *start*. Under this, we can either make a copy of $\mathcal{Q}_f$, sorted by *end*, and then use optFS or simply handle the $P^R_{\ell,i}$ and $\mathcal{Q}_f$ pair the way the typical partition-based strategy does in Algorithm 4. For our experiments, we selected the second approach to avoid the online sorting costs.

We further enhance the join computation in Lines 6, 7 and 9 by utilizing the $compfirst$ and $complast$ flags of the queries. To this end, we first have to include Lines 1–3 and Lines 24–28 from Algorithm 4. With these flags in place, we split each $\mathcal{Q}_f$ set into two subsets; subset $\mathcal{Q}^{FF}_f$ contains all queries q with both $compfirst[q]$ and $complast[q]$ set to *FALSE*, while subset $\mathcal{Q}^{rest}_f$ contains the rest. Under this breakdown, the joins in Lines 6 and 7 are also broken into $P^O_{\ell,i} \bowtie \mathcal{Q}^{FF}_f$, $P^O_{\ell,i} \bowtie \mathcal{Q}^{rest}_f$ and $P^O_{\ell,i} \bowtie \mathcal{Q}^{FF}_f$, $P^O_{\ell,i} \bowtie \mathcal{Q}^{rest}_f$. Out of the above, the two joins that include $\mathcal{Q}^{FF}_f$ are computed as cross products, following the properties of the bottom-up

5. The same point holds for the query-based and level-based, as well.

6. If $P_{\ell,i}$ is both first and last relevant for a query q, only $\mathcal{Q}_f$ has q.

ALGORITHM 5: Partition-based with sharing

Input : HINT index $\mathcal{H}$, batch of queries $\mathcal{Q}$
Output : set of all overlapping intervals, for each $q \in \mathcal{Q}$

1 **foreach** *level* $\ell = m$ **to** 0 **do** ▷ bottom-up fashion
2 **foreach** *partition* i *on level* ℓ **do**
3 $\mathcal{Q}_f \leftarrow \{q \in \mathcal{Q} : i = prefix(\ell, q.st)\}$; ▷ as first overlapping partition
4 $\mathcal{Q}_l \leftarrow \{q \in \mathcal{Q} : i = prefix(\ell, q.end)\}$; ▷ as last overlapping partition
5 $\mathcal{Q}_c \leftarrow \{q \in \mathcal{Q} : prefix(\ell, q.st) < i < prefix(\ell, q.end)\}$; ▷ fully contained partition
6 **output** $\{s : \exists (s, q) \in \mathcal{H}.P^O_{\ell,i} \bowtie \mathcal{Q}_f\}$;
7 **output** $\{s : \exists (s, q) \in \mathcal{H}.P^O_{\ell,i} \bowtie \mathcal{Q}_l\}$;
8 **output** $\{s : \exists (s, q) \in \mathcal{H}.P^O_{\ell,i} \times \mathcal{Q}_c\}$; ▷ Cross product
9 **output** $\{s : \exists (s, q) \in \mathcal{H}.P^R_{\ell,i} \bowtie \mathcal{Q}_f\}$;

approach. A similar approach can be also adopted for $\mathcal{Q}_l$ and the $P^O_{\ell,i} \bowtie \mathcal{Q}_l$ join. Finally, the $P^R_{\ell,i}$ and $\mathcal{Q}_f$ join in Line 9 naturally benefits from the above breakdown by considering the cases in Lines 13, 16, 19 and 22 from Algorithm 1, which are incorporated inside Algorithm 4 as well.

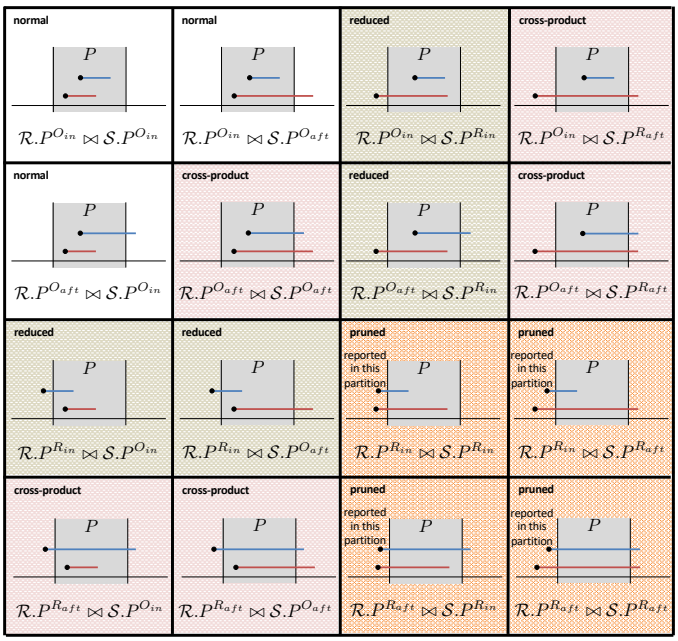

Fig. 4: Mini-joins breakdown for partition-to-partition joins

4 Processing Join Queries

We next study overlap joins under different cases depending whether the inputs are pre-indexed by HINT. Following a classification similar to [45], we consider three settings.

4.1 Unindexed Inputs

When none of the $\mathcal{R}$, $\mathcal{S}$ inputs is indexed, we can use the optFS algorithm from [34], [35] to compute the $\mathcal{R} \bowtie \mathcal{S}$ join. Next, we show how to further enhance the join performance by extending optFS towards two directions. First, the study in [34], [35] considered a space-based partitioning (referred to as "domain-based partitioning") solely as a means for processing the join in parallel. Here, we extend optFS to employ such a partitioning also for the single-threaded computation. Second, the domain partitioning in [34], [35] splits each input collection into 3 classes, termed A, B and C. Class B and C fully correspond to the $P^{R_{in}}$ and $P^{R_{aft}}$ subdivisions defined by HINT, while the A class consists of all original intervals that start inside a partition, regardless where they end, i.e., it corresponds to $P^{O_{in}} \bigcup P^{O_{aft}}$. We extend the domain-based partitioning to consider all 4 subdivisions from [11], [37], instead of the above 3.

Specifically, the unified $\mathcal{R} \bigcup \mathcal{S}$ domain is first split into k equally sized, non-overlapping stripes; each stripe holds a partition for input $\mathcal{R}$ and one for $\mathcal{S}$. Every interval $s \in \mathcal{S}$ (resp. $r \in \mathcal{R}$) is assigned to the partition of the stripe that contains $s.st$ and replicated to the partitions of all other stripes it intersects. With this partitioning in place an $\mathcal{R} \bowtie \mathcal{S}$ join is broken down into k independent partition-to-partition joins, i.e., $\mathcal{R}.P_1 \bowtie \mathcal{S}.P_1$, ..., $\mathcal{R}.P_k \bowtie \mathcal{S}.P_k$. Next, in a similar fashion to HINT, every domain partition P from each collection, is further divided into 4 subdivisions $P^{O_{in}}$, $P^{O_{aft}}$, $P^{R_{in}}$ and $P^{R_{aft}}$. With these subdivisions in place, we subsequently break down every partition-to-partition join into 16 smaller tasks, called *mini-joins*. Figure 4 illustrates this mini-joins breakdown. We next elaborate on the computation of every mini-join type:

- First, to avoid duplicate results, a join result (r, s) is reported only if at least one of the involved intervals is not a replica, i.e., if it is not contained inside a $P^{R_{in}}$ or a $P^{R_{aft}}$ subdivision. Under this premise, we never compute the 4 replica-to-replica mini-joins $\mathcal{R}.P^{R_{in}} \bowtie \mathcal{S}.P^{R_{in}}$, $\mathcal{R}.P^{R_{in}} \bowtie \mathcal{S}.P^{R_{aft}}$, $\mathcal{R}.P^{R_{aft}} \bowtie \mathcal{S}.P^{R_{in}}$ and $\mathcal{R}.P^{R_{aft}} \bowtie \mathcal{S}.P^{R_{aft}}$, shaded in orange color in Figure 4.
- The 3 original-to-original mini-joins $\mathcal{R}.P^{O_{in}} \bowtie \mathcal{S}.P^{O_{in}}$, $\mathcal{R}.P^{O_{in}} \bowtie \mathcal{S}.P^{O_{aft}}$ and $\mathcal{R}.P^{O_{aft}} \bowtie \mathcal{S}.P^{O_{in}}$ have identical complexity to the original $\mathcal{R} \bowtie \mathcal{S}$ join. Therefore, these mini-joins are evaluated as normal, i.e., using optFS from [34], [35].
- The remaining original-to-original mini-join $\mathcal{R}.P^{O_{aft}} \bowtie \mathcal{S}.P^{O_{aft}}$ differs from the previous case. By definition, an (r, s) pair of intervals in this case always overlap, regardless where their start is located inside the corresponding stripe, as they both span to the next stripe. Under this premise, $\mathcal{R}.P^{O_{aft}} \bowtie \mathcal{S}.P^{O_{aft}}$ is computed without conducting any comparisons, as a cross-product; we highlight this mini-join in pink color, in Figure 4.
- For the 4 original-to-replica mini-joins $\mathcal{R}.P^{O_{in}} \bowtie \mathcal{S}.P^{R_{in}}$, $\mathcal{R}.P^{O_{aft}} \bowtie \mathcal{S}.P^{R_{in}}$, $\mathcal{R}.P^{R_{in}} \bowtie \mathcal{S}.P^{O_{in}}$ and $\mathcal{R}.P^{R_{in}} \bowtie \mathcal{S}.P^{O_{aft}}$, a simplified (reduced) version of optFS can be used. As every replica interval inside $P^{R_{in}}$ starts in a preceding stripe, optFS only conducts forward scans to the $P^{O_{in}}$ or $P^{O_{aft}}$ subdivisions from the other input; no forward scans are needed for the replica intervals. In addition, if the grouping optimization of optFS is activated, the entire $P^{R_{in}}$ is used as a group. We highlight this mini-join type in tan color, in Figure 4.
- Each interval in $P^{R_{aft}}$ spans the entire range of the corresponding domain stripe. Such intervals overlap by definition with all intervals from the other input that start inside the same stripe, i.e., the intervals in the $P^{O_{in}}$ and $P^{O_{aft}}$ subdivisions. So, the 4 original-to-replicas mini-joins $\mathcal{R}.P^{O_{in}} \bowtie \mathcal{S}.P^{R_{aft}}$, $\mathcal{R}.P^{O_{aft}} \bowtie \mathcal{S}.P^{R_{aft}}$, $\mathcal{R}.P^{R_{aft}} \bowtie \mathcal{S}.P^{O_{in}}$ and $\mathcal{R}.P^{R_{aft}} \bowtie \mathcal{S}.P^{O_{aft}}$ are cross-products and we color them in pink in Figure 4.

ALGORITHM 6: HINT-join

Input : HINT indices $\mathcal{H}_\mathcal{R}$, $\mathcal{H}_\mathcal{S}$
Output : set of all overlapping interval pairs $(r, s) \in \mathcal{R} \times \mathcal{S}$

```
1  foreach level ℓ = max(m_R, m_S) to 0 do                    ▷ outer bottom-up
2      foreach partition H_R.P_{ℓ,i} do                       ▷ current R partition
3          j ← i;                                             ▷ same-prefix S partition on ℓ
4          output {(r,s) ∈ H_R.P_{ℓ,i} ⋈ H_S.P_{ℓ,j}};        ▷ optFS+
5          compfirst ← TRUE;
6          complast ← TRUE;
7          foreach level ℓ' = ℓ + 1 to 0 do                   ▷ inner bottom-up
8              j ← j ÷ 2;                                     ▷ same-prefix partition on ℓ'
9              if compfirst and complast then
10                 output {(r,s) ∈ H_R.P_{ℓ,i} ⋈ H_S.P_{ℓ,j}}; ▷ optFS+
11             else
12                 output {(r,s) ∈ H_R.P^O_{ℓ,i} × H_S.P_{ℓ,j}};
                       ▷ Cross-product
13             if j mod 2 = 0 then                            ▷ last bit of j is 0
14                 compfirst ← FALSE;
15             if j mod 2 = 1 then                            ▷ last bit of j is 1
16                 complast ← FALSE;
17     foreach partition H_S.P_{ℓ,i} do                       ▷ current S partition
18         j ← i;                                             ▷ same-prefix R partition on ℓ
19         compfirst ← TRUE;
20         complast ← TRUE;
21         foreach level ℓ' = ℓ + 1 to 0 do                   ▷ inner bottom-up
22             j ← j ÷ 2;                                     ▷ same-prefix partition on ℓ'
23             if compfirst and complast then
24                 output {(r,s) ∈ H_S.P_{ℓ,i} ⋈ H_R.P_{ℓ,j}}; ▷ optFS+
25             else
26                 output {(r,s) ∈ H_S.P^O_{ℓ,i} × H_R.P_{ℓ,j}};
                       ▷ Cross-product
27             if j mod 2 = 0 then                            ▷ last bit of j is 0
28                 compfirst ← FALSE;
29             if j mod 2 = 1 then                            ▷ last bit of j is 1
30                 complast ← FALSE;
```

4.2 Both Inputs Indexed

For the second setting, we assume that the input collections $\mathcal{R}, \mathcal{S}$ are indexed by the $\mathcal{H}_\mathcal{R}, \mathcal{H}_\mathcal{S}$ HINT indices, respectively, on the same domain; note that the index hierarchies need *not* to be of the same height. Next, we devise a novel algorithm termed HINT-join, which concurrently scans the two index hierarchies in a bottom-up fashion, and joins pairs of partitions as unindexed inputs. Specifically, let $\mathcal{H}_\mathcal{R}.P_{\ell,i}$ be the current partition from input $\mathcal{R}$ on level ℓ of the $\mathcal{H}_\mathcal{R}$ HINT. We join $\mathcal{H}_\mathcal{R}.P_{\ell,i}$ with the i-th partition from $\mathcal{H}_\mathcal{S}$ on the same level, i.e., $\mathcal{H}_\mathcal{S}.P_{\ell,i}$ and with the partition sharing the same prefix to i on higher levels. In contrast we ignore the rest of the $\mathcal{S}$ partitions on levels below ℓ to avoid producing duplicate results. The case of the current partition being from $\mathcal{H}_\mathcal{S}$ is symmetric, but we also ignore the $\mathcal{H}_\mathcal{S}.P_{\ell,i} \bowtie \mathcal{H}_\mathcal{R}.P_{\ell,i}$ join to avoid duplicate results.

Algorithm 6 illustrates the pseudocode of HINT-join. The algorithm combines two bottom-up traversal operations. The outer bottom-up (`for-loop` in Line 1) concurrently ascends both index hierarchies. For this purpose, the `for-loop` runs from the highest value between $m_\mathcal{R}$ and $m_\mathcal{S}$ to 0.[7] For each level ℓ, HINT-join accesses every partition $P_{\ell,i}$ from both hierarchies executing the two independent `for-loops` in Lines 2–16 and 17–30. We detail the first loop for the current partition $\mathcal{H}_\mathcal{R}.P_{\ell,i}$. We use the variable j to denote the partition from $\mathcal{H}_\mathcal{S}$ with the same prefix which will be joined with $\mathcal{H}_\mathcal{R}.P_{\ell,i}$. Initially for level ℓ, $j = i$ (Line 3) and therefore, $\mathcal{H}_\mathcal{R}.P_{\ell,i}$ is joined with $\mathcal{H}_\mathcal{S}.P_{\ell,i}$ in Line 4. Then, in the `for-loop` of Lines 7–16, HINT-join climbs $\mathcal{H}_\mathcal{S}$ in a bottom-up fashion (inner bottom-up) starting from the next level $\ell' = \ell + 1$. The algorithm joins current partition $\mathcal{H}_\mathcal{R}.P_{\ell,i}$ with the j-th partition on level ℓ' which shares the same prefix to i. This partition can be easily determined by iteratively dividing j by 2 on each level (Line 8). To accelerate this join operation, the algorithm takes advantage of HINT's bottom-up principle. Similar to Algorithm 1, we maintain the $compfirst$ and $complast$ flags by checking the last bit of j. Initially both flags are set to *TRUE*, and the HINT-join computes $\mathcal{H}_\mathcal{R}.P_{\ell,i} \bowtie \mathcal{H}_\mathcal{S}.P_{\ell',j}$. However, after both flags are switched to *FALSE*, we no longer need to consider the replicas of the current $\mathcal{R}$ partition and the join is reduced to the $\mathcal{H}_\mathcal{R}.P^O_{\ell,i} \times \mathcal{H}_\mathcal{S}.P_{\ell',j}$ cross-product (Line 12).

For the partition-to-partition joins computed by HINT-join, we can adopt the same approach as the extended optFS from the previous section because the input partitions are unindexed. In practice, since the HINT partitions usually contain significantly fewer intervals than the $\mathcal{R}$, $\mathcal{S}$ inputs, the domain-based partitioning is no longer needed but the mini-joins breakdown can be directly applied when the subdivisions are already in place for $\mathcal{H}_\mathcal{R}, \mathcal{H}_\mathcal{S}$. Regarding the remaining optimizations, out tests showed that the extended optFS is typically reduced to either an unoptimized FS or gFS, with the grouping optimization activated.

4.3 One Input Indexed

For the last setting, we assume without loss of generality that input collection $\mathcal{S}$ is indexed by HINT, while $\mathcal{R}$ is unindexed. In this case, there are essentially three alternatives to evaluate $\mathcal{R} \bowtie \mathcal{S}$. First, we can completely ignore the existing index, and evaluate the join using the extended optFS from Section 4.1. Second, we can build a temporary HINT on $\mathcal{R}$ and apply the HINT-join algorithm from Section 4.2. Third, we can evaluate the join in an indexed nested-loops fashion, where we issue a selection query to the indexed input $\mathcal{S}$ (also known as inner), for each interval in the unindexed input $\mathcal{R}$ (also known as the outer). To enhance the performance of this third approach, we treat $\mathcal{R}$ as a batch of queries and rely on the processing strategies discussed in Section 3 to efficiently process the issued selection queries.

5 Experimental Analysis

Finally, we present the results of our experimental analysis. We implemented all batch strategies for selection queries and join methods in `C++`, compiled using `gcc` (v4.8.5) with flags `-O3`, `-mavx` and `-march=native` activated. [8] Our experiments ran on an Intel(R) Xeon(R) CPU E5-2630 v4 at 2.20GHz with 384GBs of RAM, running AlmaLinux 8.5.

5.1 Setup

All methods using index were developed on top of the *subs+sort* version of HINT [11], which employs the *subdivisions* and *sorting* optimizations. We activated also the *skewness & sparsity* and the *cache misses* optimizations. However,

7. The highest m value corresponds to the tallest HINT hierarchy.

8. Source code available in https://github.com/pbour/batch_hint/.

TABLE 3: Characteristics of test 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 [sec]	31,507,200	461,829,284	31,768,287	283,356,410
Min duration [sec]	1	1	1	1
Max duration [sec]	31,406,400	461,815,512	2,148,385	59,468,008
Avg. duration [sec]	2,201,320	33,206,300	758	15
Avg. duration [%]	6.98	7.19	0.0024	0.000005

TABLE 4: Parameters of synthetic datasets

parameter	**values** (defaults in **bold**)
Domain length	32M, 64M,**128M**, 256M, 512M
Cardinality	**10M**, 50M, 100M, 500M, 1B
α (interval length)	1.01, 1.1, **1.2**, 1.4, 1.8
σ (interval position)	10K, 100K, **1M**, 5M, 10M

in line with [37], we dropped the *storage* optimization; i.e., we save each interval as a $\langle id, st, end \rangle$ triplet in the index, to ensure the best performance on selection queries for all basic relationships in Allen's Algebra [46]. Lastly, similar to previous works, all data, input collections, indices and the queries reside in main memory, while the test workload accumulates an XOR of result ids for selections and the sum of an XOR between the ids of result pair, for joins.

We experimented with 4 collections of real intervals, which have been also used in previous works; Table 3 summarizes their characteristics. BOOKS contains the periods of time in 2013 when books were lent out by Aarhus libraries (https://www.odaa.dk). WEBKIT 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 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 [47] records time periods of power usage from households in Austria and Italy from January 2010 to October 2014. Collections BOOKS and WEBKIT contain around 2M, long on average, intervals each; TAXIS and GREEND have over 100M short intervals. To build a HINT index for each dataset, we set parameter m using the cost model and the analysis in [11], i.e., 10 for BOOKS, 12 for WEBKIT and 17 for TAXIS, GREEND.

To showcase the effect of the input characteristics in batch processing, we also generated synthetic collections as in [11]. Table 4 summarizes the construction parameters and their default values. The datasets domain ranges from 32M to 512M while their cardinality, from 10M to 1B. The interval length follows a zipfian distribution, controlled by parameter α; a small value of α results in most intervals being relatively long, while with a large value the great majority of intervals have length 1. The middle point of every interval is positioned according to a normal distribution centered at the middle point of the domain. We control this position using the deviation parameter σ; the greater the value of σ, the more spread the intervals are in the domain.

5.2 Selection Queries

We start off with selection queries. To evaluate the batch processing strategies, we study their total execution cost incurred for the entire query batch, which includes the sorting costs whenever employed.[9] We ran queries uniformly distributed in the domain, varying (1) their selectivity, in terms of their extent as a percentage of the domain inside the $\{0.01\%, 0.05\%, 0.1\%, 0.5\%, 1\%\}$ range, and (2) the size of the query batch inside {1K, 5K, 10K, 50K, 100K}. In each test, we vary one of the above parameters while fixing the other to its default value; 0.1% of the domain, for the query extend; 10K for the batch size.

Figure 5 reports the total execution time for each strategy while Figure 6 reports on advanced statistics, namely the number of cache misses at the first (L1) and last (LLC) cache level, and the number of issued instructions; due to lack of space, we include the numbers only for the default query setting. In the first row of plots in Figure 5, we vary the selectivity of the queries, and in the second, the size of the query batch. We consider the query-based strategy without sorting powered by HINT as our baseline.[10] As the first observation, the tests confirm our intuition in Section 3.1; examining the queries in the batch sorted by their start will enhance the performance, due to reducing the number of horizontal jumps and improving the cache locality. Indeed, the query-based strategy with this sorting clearly outperforms the baseline query-based without, in all cases. The performance gain is more pronounced in TAXIS and GREEND, where intervals are typically stored at the bottom levels, rendering the horizontal jumps more impactful. Our findings are supported by the cache misses stats in Figure 6 especially at LLC, showing that query-based without sorting will issue more requests to the main memory. Under this prism, we test level-based and partition-based (both variants) with the query sorting always activated.

In addition, the experiments show the benefits of batch processing and the advantage of our proposed level-based and partition-based advanced strategies over query-based with sorting. We observe that the performance gain is in practice larger in case of BOOKS and WEBKIT, compared to TAXIS and GREEND, because of the length of the contained intervals (see Table 3). The intervals in BOOKS and WEBKIT are stored at the higher levels of the index due to their significantly large length. Consequently, the impact of the vertical jumps is more pronounced in these datasets. As a result, level-based has almost identical total times to query-based with sorting on TAXIS and GREEND, while partition-based is always faster, because additional horizontal jumps are avoided by depleting all queries relevant to a partition before moving to the next, at the current level. Again, our findings are backed up by the advanced stats in Figure 6. For partition-based, partition subdivisions often reside in L1, allowing to fast scan them and answer the relevant queries.

When access sharing is employed, the performance of partition-based is significantly enhanced; in fact this partition-based variant is up to one order of magnitude faster than the typical partition-based. The performance gap is larger for BOOKS, WEBKIT in all tests and for all datasets

9. To deal with latency, systems employ a waiting timeout for defining a batch. When the waiting time exceeds this threshold, the batch is executed regardless its size. Under this premise, the system executes the previous batch while waiting for the next, and so, we can completely ignore the waiting time in our experiments.

10. Our analysis in [11], [37] already showed the advantage of a HINT-powered query-based strategy against competitive indexing.

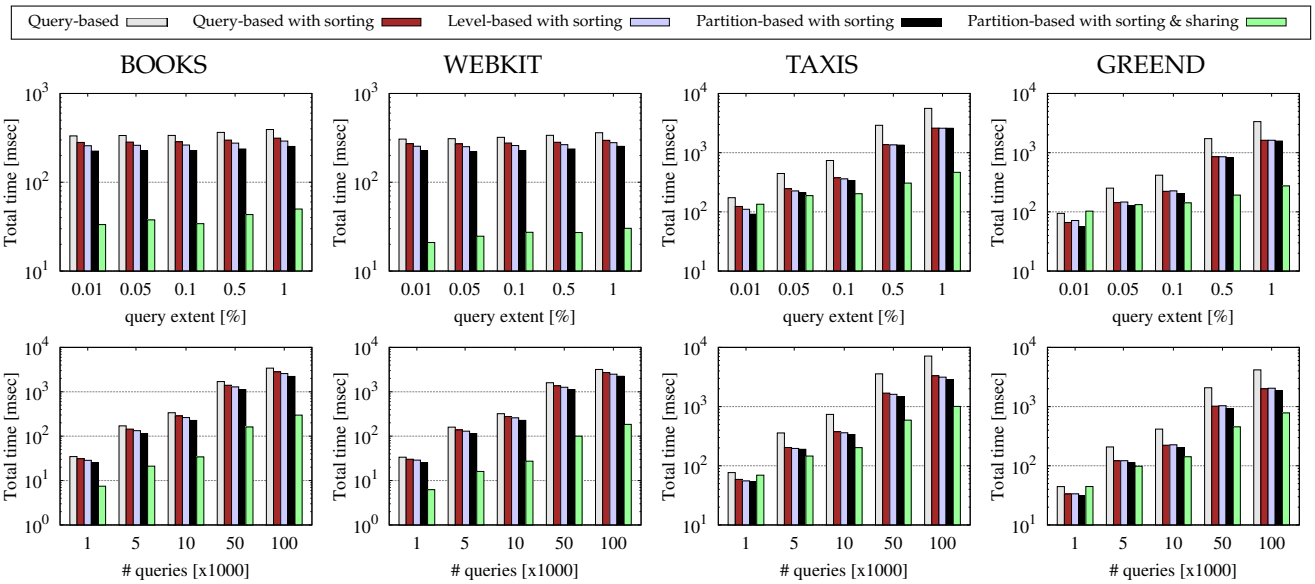

Fig. 5: Batch processing selection queries (real datasets)

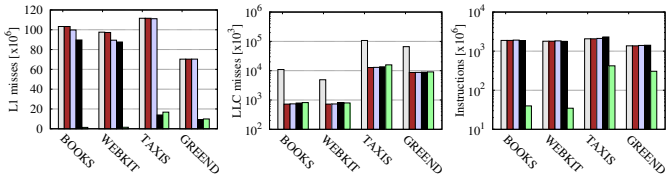

Fig. 6: Advanced stats; defaults 0.1% extent, 10K queries

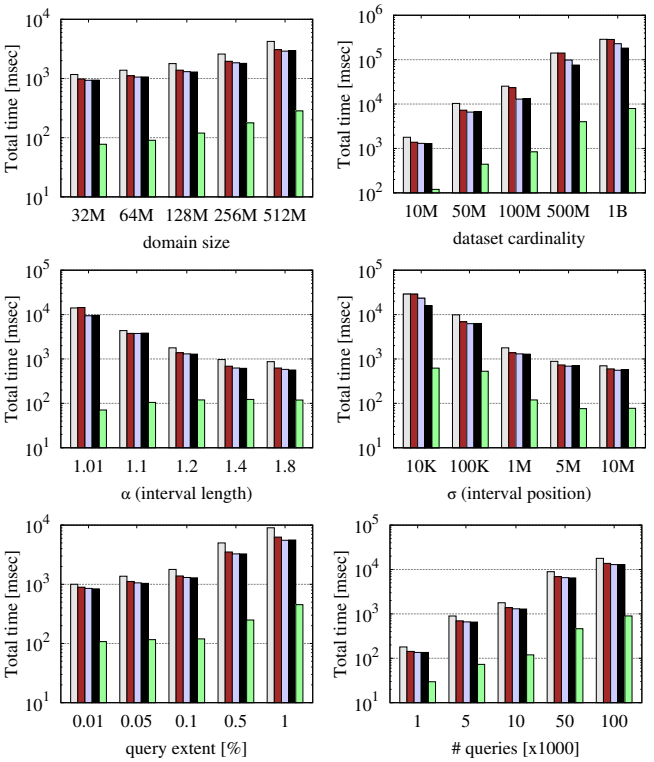

Fig. 7: Batch processing selection queries (synthetic datasets)

when the selectivity is high or the batch contains a lot of queries. Besides ensuring that each partition is scanned once, this is also due to the large number of cross products conducted in these case by the access sharing approach. These two factors drastically reduce both the number of issued instructions and the cache misses in L1, justifying the performance advantage over the original partition-based and all other strategies; see Figure 6.

To highlight the impact of batch processing, Figure 8 reports the percentage of the queries inside a batch that would have been executed in a serial fashion, within the total time of each strategy; the lower this percentage, the more queries are positively affected by batch processing. The figure clearly shows both the benefit of batch processing selection queries over a serial execution (with or without sorting) and the advantage of the partition-based strategy with access sharing activated. An exception arises for a very small batch (1K queries) and TAXIS, WEBKIT, where using optFS to join a subdivision with all its relevant queries does not pay off; this finding is in line with the second row in Figure 5. As expected this benefit grows for all strategies as the batch size (i.e., number of contained queries) increases.

Regarding the impact of the experimental parameters, all strategies are slowed down (1) when increasing the query extent as the queries become less selective and so, more time-consuming with larger result sets, and (2) when increasing the batch size, as more queries are evaluated. Nevertheless, partition-based with access sharing is consistently the faster strategy for all datasets and in all tests.

Lastly, Figure 7 reports on the synthetic datasets. Parameter m is again set using the model in [11]. The plots unveil similar trends to Figure 5. As expected the domain size, the dataset cardinality, the query extent and the batch size, all negatively affect the performance of the strategies. 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. In contrast, when α grows, the intervals become shorter, so the performance of all strategies 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.

5.3 Join Queries

We continue with join queries. Our analysis considers two setups, depending on whether input collections are already

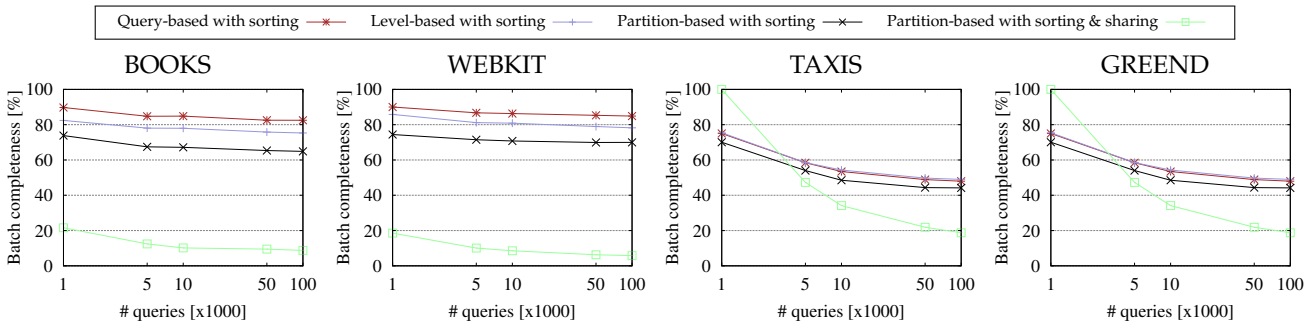

Fig. 8: Impact of batch processing - lower numbers better; default query extent 0.1%

indexed (i.e., to answer other queries such as selections), or the inputs are unindexed. To assess the performance of each join method, we measured its total execution time which includes sorting, indexing and partitioning costs, wherever applied. We ran a series of interval joins using uniformly sampled subsets of each real collection as the outer input R and the entire dataset as the inner S; for each setting, the $|R|/|S|$ ratio varies inside the $\{0.25, 0.5, 0.75, 1\}$ set.[11]

5.3.1 Unindexed Inputs

We compare the state-of-the-art optFS method for joining unindexed inputs from [34], [35] (recapped in Section 2.2) against our extension in Section 4.1; we denote the latter as optFS+.[12]Our extension method first splits the domain into equally sized, non-overlapping stripes and constructs the O_{in}, O_{aft}, R_{in}, R_{aft} subdivisions inside a stripe, for each dataset. Then, optFS+ applies the mini-joins breakdown in Figure 4 and utilizes optFS to compute the mini-joins. Recall that optFS samples the input subdivisions to decide which optimizations will be activated for each mini-join, as proposed in [35] for the entire input datasets.

Figure 9 investigates the best granularity for the partitioning of optFS+; the plots show a breakdown of the total time. Partitioning time measures the cost of splitting the domain into stripes, constructing the 4 subdivisions inside each stripe and computing the decomposed layout (if activated). Sorting is required in order to use the plane-sweep based approach of optFS. Indexing time measures the cost of building the bucketing index (if optimization activated). The cost of determining the groups for the grouping optimization (if activated) is included in the joining time. As the key take away of Figure 9, we observe that 100 stripes (partitions) incur the best performance for optFS+ on datasets with long intervals, such as BOOKS and WEBKIT, while on the datasets with short intervals such as TAXIS and GREEND, the best performance is exhibited for 100K stripes. Hence, we fix the partitioning for the next tests accordingly. Note that the stiff decrease in time for TAXIS from 10K to 100K is due to optFS+ switching off the grouping, bucket indexing and decomposed layout optimizations. These optimizations no longer pay off as the capacity of the TAXIS partitions significantly drops.

Finally, Figure 10 compares the performance of optFS+ against the optFS state-of-the-art. The results confirm that the domain-based partition with the mini-joins breakdown further accelerates the join computation, even under a single-thread setting; recall that [34], [35] considered this optimization only for a multi-threaded setting. optFS+ is the best method for joining undindexed inputs in the majority of our tests. Only for inputs with extremely short intervals, such as the GREEND, we observe that the partitioning does not pay off. Nevertheless, the join in these cases is already significantly cheaper compared to the other datasets.

5.3.2 Indexed Inputs

When indexing already exists in the input collections, we consider two approaches for computing an overlap join. The first is a typical index-based nested-loops approach, where $\mathcal{R} \bowtie \mathcal{S}$ is evaluated as a series of selection queries. Specifically, for each interval in the outer input, e.g., $r \in \mathcal{R}$, we issue the $\sigma_{[r.st,r.end]}(\mathcal{S})$ selection query in the inner input $\mathcal{S}$, to determine all s intervals that overlap r. We evaluate all these selection queries in a single batch utilizing the partition-based approach with sorting and sharing, which was shown to have the best performance in Section 5.2. For completeness purposes, we additionally consider OMJ^i from [32] which also follows an index-based nested-loops approach. However, instead of utilizing an interval index, it uses a typical relational index, i.e., a B$^+$-tree[13], to efficiently evaluate the $\sigma_{[r.st \leq s.st \leq r.end]}(\mathcal{S})$ selection for each $r \in \mathcal{R}$ and the $\sigma_{[s.st < r.st \leq s.end]}(\mathcal{R})$ selection for each $s \in \mathcal{S}$. Finally, the second join approach is the HINT-join method described in Section 4.2 which uses the HINT indices on both inputs.

Figure 11 reports the results of our tests. For the index-based nested-loops approach, the largest input is typically selected as inner. To confirm this rule, we experimented with either $\mathcal{R}$ or $\mathcal{S}$ ($|\mathcal{R}| \leq |\mathcal{S}|$) as the inner input. Our analysis confirms the above rule only for BOOKS and WEBKIT that contain long intervals; using the HINT on the larger input $\mathcal{S}$ is always faster than using the index on $\mathcal{R}$. However, for the TAXIS and GREEND collections that contain short intervals, it is more beneficial to probe the HINT on the smaller $\mathcal{R}$. When comparing the index-based nested-loops methods to each other, OMJ^i is competitive only if the input collections contain short intervals; in contrast, for BOOKS and WEBKIT, OMJ^i is usually two times slower than the best batch processing on HINT. Regardless, we observe that none of the indexed-based nested-loops methods can compete with HINT-join, which takes full advantage of the interval indexing on both inputs. Finally, by juxtaposing

11. We also tested disjoint subsets observing similar behavior.

12. We omit the OMJ method in [32], which adopts a merge-join evaluation, as it was shown to have similar or even worse performance than bgFS [34], a preliminary and slower version of optFS, which only uses the grouping and bucket indexing optimizations.

13. We used the implmentation from the TLX library [48].

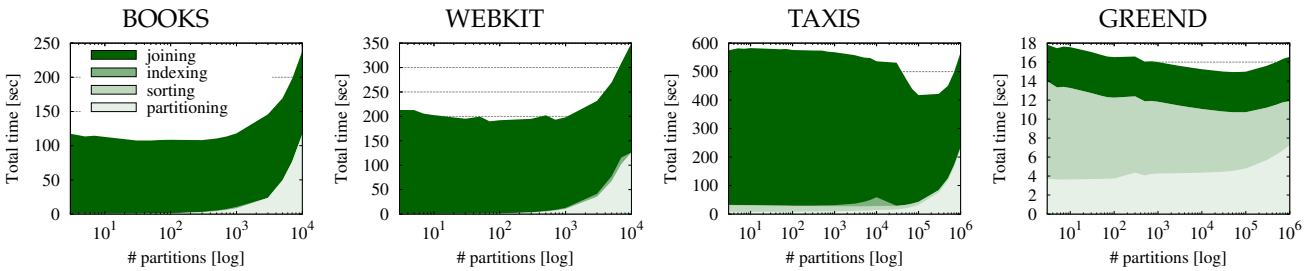

Fig. 9: Determining the optimal partitioning granularity for optFS+, time breakdown

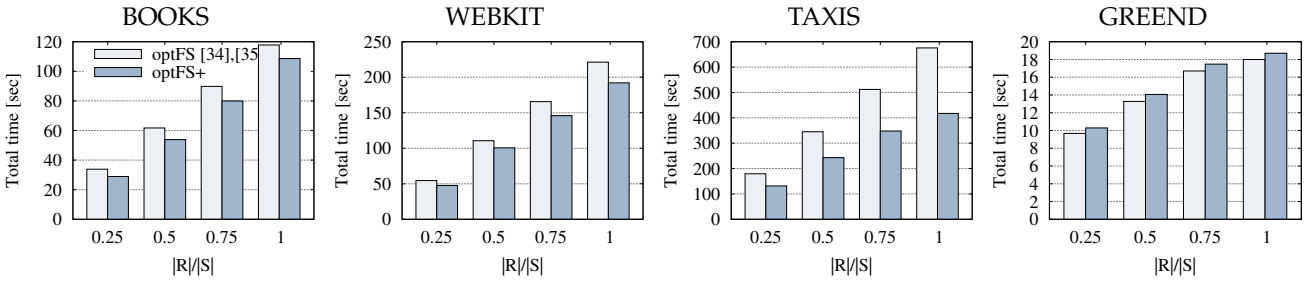

Fig. 10: Joining unindexed inputs

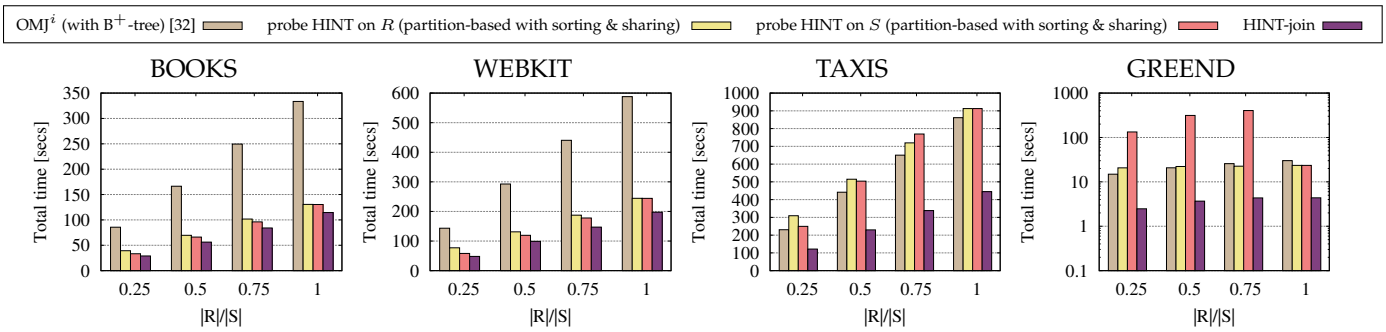

Fig. 11: Joining indexed inputs

Figures 10 with 11, we observe that optFS+ and HINT-join have comparable performance with the exception of GREEND where HINT-join is the best method.

6 Related Work

We discuss previous work on indexing and joining intervals, besides HINT and optFS, respectively, covered in Section 2.

Indexing intervals. A simple data structure for intervals is a *1D-grid*, which divides the domain into k pair-wise disjoint, partitions $P_1, P_2, \ldots, P_k$. Input intervals are assigned to all partitions they intersect and selection queries q, obtain results by accessing all partition P_i that overlap with q. The reference value [49] can be adapted for result deduplication.

The *interval tree* [8] offers optimal worst-case space and time guarantees. The tree divides the input 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. To answer a selection range query, the interval tree is traversed top-down comparing the center represented on each node to the query range. A relational interval tree RI-tree for disk-resident data was proposed in [29]. Another binary search tree for intervals is the *segment tree* [5], which however was designed for stabbing (or point) queries where the goal is to determine the intervals that contain a specific value.

Other solutions for indexing intervals are the *timeline index*, the *period index* and the *RD-index*. The timeline index [50] is a general-purpose access method for temporal (versioned) data, implemented as SAP-HANA tables. A table, called the event list, stores a $\langle time, id, isStart \rangle$ triple for the endpoints of all intervals, where $time$ is either the start or end of an interval, specified accordingly by the boolean $isStart$ flag. In addition, at certain timestamps, called *checkpoints*, the entire set of active objects is materialized, i.e., those with an interval that contain the checkpoint. Selection queries $q = [q.st, q.end]$ are evaluated by comparing the contents of the closest checkpoint before $q.st$ and the entries in the event list after the checkpoint against the query range. Period [9] and RD-index [10] split the domain into coarse partitions, then subdivide each partition hierarchically to organize intervals by position and duration, optimized for range and duration queries. The RD-index [10] essentially improves upon the period index by supporting arbitrary distributions of interval durations and allowing to index the intervals either first by duration or time. Moreover, RD-index does not replicate intervals, yielding a smaller memory footprint and better query performance.

The experiments in [11], [37] showed that HINT outperforms the above methods for selection queries. Moreover,

HINT exhibits low space complexity at a competitive building time. Thus, our study builds upon HINT as the state-of-the-art interval indexing solution in main memory.

Joining intervals Early works [23], [24] on interval joins relied on nested-loops (unsorted inputs) or sort-merge join (sorted inputs), introducing specialized data structures for append-only data. To reduce join costs, index-based and partitioning-based solutions emerged. Enderle et al. [31] leveraged two RI-trees [29] while Zhang et al. [30] employed an extension of the multi-version B-tree [51]. Luo et al. [52] proposed *O2iJoin* which utilizes a flat two-level index, where the first level organizes the indexed input in a sorted array and the second level contains inverted lists that approximate the nesting structure of the period timestamps. Recently, Dignös et al. [32] reformulated $\mathcal{R} \bowtie \mathcal{S}$ overlap joins as the union (with duplicate elimination) of two range queries $\sigma_{[r.st \leq s.st \leq r.end]}(\mathcal{S})$ and $\sigma_{[s.st < r.st \leq s.end]}(\mathcal{R})$. For unindexed inputs, the proposed OMJ method adopts a sort-merge join approach while for indexed inputs, OMJ^i evaluates the range queries using two B$^+$-trees, which index the starting point of the intervals.

Soo et al. [25] proposed a partitioning-based approach which first splits the domain into disjoint ranges and then, assigns each interval to the partition of the last domain range it overlaps. To evaluate the join, the partitions are processed sequentially from last to first; after the last pair of partitions are processed, the intervals which overlap the previous domain range are migrated to the next join. This way data replication is avoided. Dignös et al. [27] proposed the *Overlap Interval Partitioning* (OIP) join which divides the inputs into groups of intervals with similar endpoints, maximizing the percentage of matching objects per partition. The *Disjoint Interval Partitioning* (DIP) algorithm proposed in [28] creates disjoint partitions, each containing non-overlapping intervals. To compute the overlap join a sort-merge approach without backtracking is employed.

For unindexed inputs, methods that build upon plane-sweep have been also proposed. Similar to sort-merge join evaluation, plane-sweep based methods require the two input collections to be sorted, but they can guarantee at most $|\mathcal{R}| + |\mathcal{S}|$ comparisons to produce the results. The *Endpoint-based Interval* (EBI) algorithm and its lazy version LEBI [33], [36] extend the timeline index core idea utilizing the same event list. To compute the join, EBI concurrently scans the input event lists accessing their entries in increasing global order of their sorting key (i.e., the endpoint), simulating a "sweep line" that stops at each endpoint from either input $\mathcal{R}$ or $\mathcal{S}$. At each position of the sweep line, the algorithm keeps track of the active (open) intervals using a gapless hash map optimized for sequential reads.

The tests in [33], [36] and [34], [35] showed the advantage of the plane-sweep based evaluation for overlap joins on unindexed inputs. Between EBI/LEBI [33], [36] and optFS [34], [35], the latter achieves competitive or better performance, without any specialized data structure. In addition, the tests in [32] showed that OMJ performs comparably to bgFS [34], a preliminary and slower version of optFS. Hence, our study adopts optFS as the state-of-the-art join method for unindexed inputs. For indexed inputs, we build upon HINT and consider as a competitor, OMJ^i from [32].

7 Conclusions

We studied two fundamental querying operations on intervals building on the state-of-the-art in-memory index for intervals, HINT. For the efficient batch processing of selection (range) queries, we proposed two novel strategies termed level-based and partition-based, which evaluate all queries for an index level before moving to the next. In addition, we extended partition-based to guarantee that each index partition is scanned only once for all relevant queries. Our tests showed that our strategies always outperform the serial execution of the queries. We also study overlap joins anew across the entire spectrum of different setups. For unindexed inputs, we enhanced the state-of-the-art algorithm optFS with effective partitioning employed in HINT, while for indexed inputs, we devise a new join algorithm termed HINT-join, which concurrently scans the HINT indices, joining partition pairs with optFS.

In the future, we plan to study the parallel and distributed processing of the queries. For our batch processing strategies, we will investigate how to schedule the scanning of HINT partitions to the available threads in multi-core environments, and how to distributively store and process these partitions in the available machines. Another interesting direction is integrating HINT in OpenMLDB as its time-travel data structure to enable faster retrieval of relevant intervals; that would also enable our batch-processing techniques into OpenMLDB's Scale-OIJ [7] to reduce redundant computations and improve cache efficiency.

References

[1] R. T. Snodgrass and I. Ahn, "Temporal databases," *Computer*, vol. 19, no. 9, pp. 35–42, 1986.

[2] M. H. Böhlen, A. Dignös, J. Gamper, and C. S. Jensen, "Temporal data management - an overview," in *eBISS*, 2017, pp. 51–83.

[3] N. N. Dalvi and D. Suciu, "Efficient query evaluation on probabilistic databases," in *VLDB*, 2004, pp. 864–875.

[4] P. Samarati and L. Sweeney, "Generalizing data to provide anonymity when disclosing information (abstract)," in *ACM PODS*, 1998, p. 188.

[5] M. de Berg, O. Cheong, M. J. van Kreveld, and M. H. Overmars, *Computational geometry: algorithms and applications, 3rd Edition*. Springer, 2008.

[6] A. Awad, R. Tommasini, S. Langhi, M. Kamel, E. D. Valle, and S. Sakr, "D^{2}IA: user-defined interval analytics on distributed streams," *Information Systems*, vol. 104, p. 101679, 2022.

[7] H. Zhang, X. Zeng, S. Zhang, X. Liu, M. Lu, and Z. Zheng, "Scalable online interval join on modern multicore processors in openmldb," in *IEEE ICDE*, 2023, pp. 3031–3042.

[8] H. Edelsbrunner, "Dynamic rectangle intersection searching," Institute for Information Processing, Technical University of Graz, Austria, Tech. Rep. 47, 1980.

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

[10] M. Ceccarello, A. Dignös, J. Gamper, and C. Khnaisser, "Indexing temporal relations for range-duration queries," in *SSDBM*, 2023, pp. 3:1–3:12.

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

[12] T. K. Sellis, "Multiple-query optimization," *ACM Trans. Database Syst.*, vol. 13, no. 1, pp. 23–52, 1988.

[13] M. Eslami, V. Mahmoodian, I. Dayarian, H. Charkhgard, and Y. Tu, "Query batching optimization in database systems," *Comput. Oper. Res.*, vol. 121, p. 104983, 2020.

[14] Y. Tu, M. Eslami, Z. Xu, and H. Charkhgard, "Multi-query optimization revisited: A full-query algebraic method," in *IEEE Big Data*, 2022, pp. 252–261.

[15] A. Papadopoulos and Y. Manolopoulos, "Multiple range query optimization in spatial databases," in *ADBIS*, 1998, pp. 71–82.

[16] Y. Chen and J. M. Patel, "Efficient evaluation of all-nearest-neighbor queries," in *IEEE ICDE*, 2007, pp. 1056–1065.

[17] F. M. Choudhury, J. S. Culpepper, Z. Bao, and T. Sellis, "Batch processing of top-*k* spatial-textual queries," *ACM Trans. Spatial Algorithms Syst.*, vol. 3, no. 4, pp. 13:1–13:40, 2018.

[18] W. Le, A. Kementsietsidis, S. Duan, and F. Li, "Scalable multi-query optimization for SPARQL," in *ICDE*, 2012, pp. 666–677.

[19] L. Zervakis, V. Setty, C. Tryfonopoulos, and K. Hose, "Efficient continuous multi-query processing over graph streams," in *EDBT*, 2020, pp. 13–24.

[20] L. Li, M. Zhang, W. Hua, and X. Zhou, "Fast query decomposition for batch shortest path processing in road networks," in *IEEE ICDE*, 2020, pp. 1189–1200.

[21] S. Ding, J. Attenberg, R. Baeza-Yates, and T. Suel, "Batch query processing for web search engines," in *WSDM*, 2011, pp. 137–146.

[22] J. Mackenzie and A. Moffat, "Index-based batch query processing revisited," in *ECIR*, 2023, pp. 86–100.

[23] A. Segev and H. Gunadhi, "Event-join optimization in temporal relational databases," in *VLDB*, 1989.

[24] H. Gunadhi and A. Segev, "Query processing algorithms for temporal intersection joins," in *IEEE ICDE*, 1991, pp. 336–344.

[25] M. D. Soo, R. T. Snodgrass, and C. S. Jensen, "Efficient evaluation of the valid-time natural join," in *IEEE ICDE*, 1994.

[26] I. Sitzmann and P. J. Stuckey, "Improving temporal joins using histograms," in *DEXA*, 2000.

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

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

[29] H. Kriegel, M. Pötke, and T. Seidl, "Managing intervals efficiently in object-relational databases," in *VLDB*, 2000, pp. 407–418.

[30] D. Zhang, V. J. Tsotras, and B. Seeger, "Efficient temporal join processing using indices," in *IEEE ICDE*, 2002.

[31] J. Enderle, M. Hampel, and T. Seidl, "Joining interval data in relational databases," in *ACM SIGMOD*, 2004.

[32] A. Dignös, M. H. Böhlen, J. Gamper, C. S. Jensen, and P. Moser, "Leveraging range joins for the computation of overlap joins," *VLDB J.*, vol. 31, no. 1, pp. 75–99, 2022.

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

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

[35] P. Bouros, N. Mamoulis, D. Tsitsigkos, and M. Terrovitis, "In-memory interval joins," *VLDB J.*, vol. 30, no. 4, 2021.

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

[37] G. Christodoulou, P. Bouros, and N. Mamoulis, "HINT: a hierarchical interval index for allen relationships," *VLDB J.*, vol. 33, no. 1, pp. 73–100, 2024.

[38] P. Bouros, A. Titkov, G. Christodoulou, C. Rauch, and N. Mamoulis, "HINT on steroids: Batch query processing for interval data," in *EDBT*, 2024, pp. 440–446.

[39] T. Brinkhoff, H. Kriegel, and B. Seeger, "Efficient processing of spatial joins using r-trees," in *ACM SIGMOD*, 1993.

[40] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1977.

[41] A. Nicolau, "Loop quantization: Unwinding for fine-grain parallelism exploitation," Dept. of Computer Science, Cornell University, Tech. Rep. TR85-709, October 1985.

[42] W. P. Petersen and P. Arbenz, *Introduction to Parallel Computing*. Oxford Press, 2004.

[43] G. P. Copeland and S. Khoshafian, "A decomposition storage model," in *ACM SIGMOD*, 1985.

[44] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik, "C-store: A column-oriented DBMS," in *VLDB*, 2005, pp. 553–564.

[45] N. Mamoulis, *Spatial Data Management*, ser. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[46] J. F. Allen, "Maintaining knowledge about temporal intervals," *Commun. ACM*, vol. 26, no. 11, pp. 832–843, 1983.

[47] A. Monacchi, D. Egarter, W. Elmenreich, S. D'Alessandro, and A. M. Tonello, "GREEND: an energy consumption dataset of households in italy and austria," in *SmartGridComm*, 2014, pp. 511–516.

[48] T. Bingmann, "TLX: Collection of sophisticated C++ data structures, algorithms, and miscellaneous helpers," 2018, https://github.com/tlx/tlx.

[49] J. Dittrich and B. Seeger, "Data redundancy and duplicate detection in spatial join processing," in *IEEE ICDE*, 2000, pp. 535–546.

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

[51] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, "An asymptotically optimal multiversion b-tree," *VLDB J.*, vol. 5, no. 4, pp. 264–275, 1996.

[52] J. Luo, S. Shi, G. Yang, H. Wang, and J. Li, "O2ijoin: An efficient index-based algorithm for overlap interval join," *J. Comput. Sci. Technol.*, vol. 33, no. 5, pp. 1023–1038, 2018.

Panagiotis Bouros received his diploma and doctorate degrees from the School of Electrical and Computer Engineering, National Technical University of Athens, Greece, in 2003 and 2011, respectively. Currently, he is an assistant professor at the Institute of Computer Science, Johannes University Gutenberg Mainz, Germany. Before, he held positions in Denmark, Germany and Hong Kong. His research focuses on managing complex data types and query processing.

George Christodoulou received his diploma, master and doctorate degrees from the Computer Science and Engineering Department, University of Ioannina, Greece in 2017, 2020 and 2023, respectively. Currently, he is a post-doctoral researcher at TU Delft in the Netherlands; his research interests include scalable data management and distributed transactions.

Christian Rauch has a master degree in Information Systems from the University of Göttingen, Germany. He is currently a doctorate researcher at Johannes Gutenberg University Mainz; his research focuses on temporal information retrieval, indexing and query processing.

Artur Titkov received his bachelor degree from the Institute of Computer Science, Johannes University Gutenberg Mainz, Germany. He is currently working as a Database Specialist at the LORENZ Life Sciences Group.

Nikos Mamoulis received his diploma in computer engineering and informatics in 1995 from the University of Patras, Greece, and his PhD in computer science in 2000 from HKUST. He is currently a professor at the University of Ioannina, Greece. Before, he was a faculty member at the Department of Computer Science, University of Hong Kong. His research focuses on managing and mining of complex data types.