REGULAR PAPER

In-Memory Interval Joins

Panagiotis Bouros[1] · **Nikos Mamoulis**[2] · **Dimitrios Tsitsigkos**[2] · **Manolis Terrovitis**[3]

Abstract

The interval join is a popular operation in temporal, spatial, and uncertain databases. The majority of interval join algorithms assume that input data reside on disk and so, their focus is to minimize the I/O accesses. Recently, an in-memory approach based on plane sweep (PS) for modern hardware was proposed which greatly outperforms previous work. However, this approach relies on a complex data structure and its parallelization has not been adequately studied. In this article, we investigate in-memory interval joins in two directions. First, we explore the applicability of a largely ignored forward scan (FS)-based plane sweep algorithm, for single-threaded join evaluation. We propose four optimizations for FS that greatly reduce its cost, making it competitive or even faster than the state-of-the-art. Second, we study in depth the parallel computation of interval joins. We design a non-partitioning-based approach that determines independent tasks of the join algorithm to run in parallel. Then, we address the drawbacks of the previously proposed hash-based partitioning and suggest a domain-based partitioning approach that does not produce duplicate results. Within our approach, we propose a novel breakdown of the partition-joins into mini-joins to be scheduled in the available CPU threads and propose an adaptive domain partitioning, aiming at load balancing. We also investigate how the partitioning phase can benefit from modern parallel hardware. Our thorough experimental analysis demonstrates the advantage of our novel partitioning-based approach for parallel computation.

Keywords Interval data · Join · Query processing · Plane sweep · Parallel processing · Main memory

1 Introduction

Given a 1D discrete or continuous domain, an interval is defined by a starting and an ending point in this domain. Consider for example the domain of all non-negative integers $\mathbb{N}$; two integers start, end $\in \mathbb{N}$, with start $\leq$ end define an interval $i = [\text{start}, \text{end}]$ as the subset of $\mathbb{N}$, which includes all integers x with start $\leq x \leq$ end. [1] Let R, S be two collections of intervals. The *interval join* $R \bowtie S$ is defined by all pairs of intervals $r \in R$, $s \in S$ that *intersect*, i.e., $r.\text{start} \leq s.\text{start} \leq r.\text{end}$ or $s.\text{start} \leq r.\text{start} \leq s.\text{end}$.

The interval join is one of the most widely used operations in temporal databases [16]. Generally speaking, temporal databases store relations of explicit attributes that conform to a schema and each tuple carries a *validity interval*. In this context, an interval join would find pairs of tuples from two relations which have intersecting validity. For example, assume that the employees of a company may be employed at different departments during different time periods. Given the employees in Fig. 1 who have worked in departments A (red), B (blue), the interval join would find pairs of employees, whose periods of work in A and B, respectively, overlap.

Interval joins find application in other domains as well. In multidimensional spaces, an object can be represented as a set of intervals from a space-filling curve. The intervals correspond to the subsequences of points on the curve that are

✉ Panagiotis Bouros
  bouros@uni-mainz.de

  Nikos Mamoulis
  nikos@cs.uoi.gr

  Dimitrios Tsitsigkos
  dtsitsigkos@cse.uoi.gr

  Manolis Terrovitis
  mter@imis.athena-innovation.gr

1 Institute of Computer Science, Johannes Gutenberg University Mainz, Mainz, Germany

2 Department of Computer Science and Engineering, University of Ioannina, Ioannina, Greece

3 Information Management Systems Institute, Athena Research Center, Marousi, Greece

---

1 Note that the intervals in this paper are *closed*. Yet, our techniques and discussions apply on generic intervals where the begin and end sides are either open or closed.
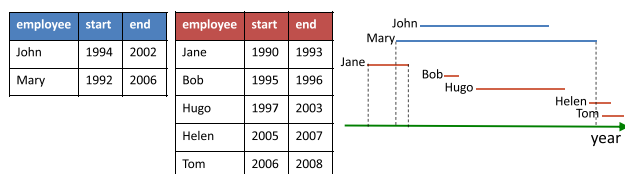
Fig. 1 Motivation example in temporal databases

included in the object. Spatial joins can then be reduced to interval joins in the space-filling curve representation [22]. The filter-step of spatial joins between sets of objects approximated by minimum bounding rectangles (MBRs) can also be processed by finding intersecting pairs in one dimension (i.e., an interval join) and verifying the intersection in the other dimension on-the-fly [2,7,36]. Another application is uncertain data management. Uncertain values are represented as intervals (which can be paired with confidence values). Thus, equi-joins on the uncertain attributes of two relations translate to interval joins [11].

Most of the previous works on interval joins [13,15,18, 32,34] assume that the input data reside on disk and their objective is to minimize I/O accesses during the join. Such a setting becomes less relevant in contemporary in-memory data management and the wide availability of parallel and distributed platforms and models. Hence, the classic *plane sweep* (PS) algorithm [31] for in-memory join evaluation has not been the focus in most of the previous work. A recent paper [29] proposed an optimized PS algorithm (taken from [2]), called Endpoint-Based Interval (EBI) Join. EBI sorts the endpoints of all intervals (from both R and S) and then *sweeps* a line which stops at each of the sorted endpoints. As the line sweeps, EBI maintains the *active sets* of intervals from R and S which intersect with the current stop point of the line to output the join results.

The work of [29] focused on minimizing the random memory accesses due to the updates and scans of the active sets. To this end, a special data structure called *gapless hash map* was proposed. However, random accesses can be overall avoided by another implementation of PS, presented in [7] for MBR (i.e., spatial) joins. We call this version *forward scan* (FS) based PS. In a nutshell, FS sweeps all intervals in increasing order of their start points. For each interval encountered (e.g., $r \in R$), FS scans forward the list of intervals from the other set (e.g., S). All such intervals having their start point before the end of r form join results with r. The cost of FS (excluding sorting) is $O(|R| + |S| + |R \bowtie S|)$, where $|R \bowtie S|$ is the number of join results.

Contributions In this work, we investigate the in-memory computation of interval joins, taking advantage of the parallel processing offered by modern multi-core hardware. Our contributions are twofold. First, we study the single-threaded computation of interval joins, by presenting four novel opti-

mizations for the FS algorithm, which greatly reduce its computational cost. In particular, optimized FS manages to produce multiple join tuples in batch at the cost of a single comparison or even output some results with zero comparisons. The performance of FS is further enhanced by careful storage of the intervals in main memory, which reduces cache misses. Overall, we achieve competitive or better performance to the state-of-the-art PS algorithm (EBI [29]), without using any special data structures.

Second, we study the in-memory parallel computation of interval joins. We investigate two approaches that differ on whether they physically partition the inputs. Our *no-partitioning* method operates in a *master-slaves* manner; the master CPU thread sweeps input intervals, while slave threads perform independent forward scans in parallel. For partitioning-based parallel processing, we first show the limitations of the hash-based partitioning framework from [29]. Then, we propose a novel, domain-based partitioning instead. Although intervals should be replicated in the domain partitions to ensure correctness, as we show, duplicate results can be avoided, therefore the partition-join jobs can become completely independent. To minimize the number of comparisons and also achieve load balancing, we break down each partition-join into five independent *mini-join* jobs with varying costs; in practice, only one of these mini-joins has the complexity of the original join problem, while the others have a significantly lower cost. We show how to schedule these mini-joins to the available CPU threads. To improve the cost balancing between the partition-joins, we also suggest an adaptive splitting approach. Finally, we present and evaluate three strategies for the partitioning phase which benefit from modern hardware. Our experimental analysis shows that the domain-based partitioning framework, after employing all the proposed optimizations, achieves high speedup with the number of threads, greatly outperforming both the hash-based partitioning framework of [29] and the no-partitioning approach.

Comparison to our previous work This article significantly extends a preliminary version of our work [5] in a number of directions. First, we design two additional optimization techniques for FS which further boost its performance. All optimizations are thoroughly evaluated, including new experiments to provide better insights. Second, we provide a rule of the thumb that decides which optimizations to apply, based on the characteristics of the join inputs. Accordingly, we devise optFS, a self-tuning version of FS, which automatically selects and applies the most appropriate optimizations. Third, we present a specialized version of FS for *interval self-joins*, i.e., when we seek overlapping pairs of intervals in a single collection. Fourth, we discuss and evaluate a new approach for parallel processing which does not physically partition the inputs. Fifth, we investigate alternative strategies for the partitioning phase of the join. Finally, we conduct new

tests to investigate the best setup for each parallel processing framework.

Outline The rest of the article is organized as follows. First, Sect. 2 discusses related work while Sect. 3 reviews in more detail plane sweep methods; EBI [29] and original FS [7]. Then, we discuss the single-threaded join evaluation. Section 4 details our optimizations for FS, Sect. 5 discusses self-joins and Sect. 6 presents our experimental analysis which demonstrates the effect of our FS optimizations. Next, we discuss the parallel computation of interval joins. Section 7 presents two novel parallel techniques, termed no-partitioning and domain-based partitioning, Sect. 8 details our strategies for parallelizing the partitioning phase and Sect. 9 presents the second part of our experiments that demonstrates the efficiency of our parallel interval join framework. Last, Sect. 10 concludes the paper.

2 Related work

We classify previous works based on the data structures they use and on the underlying architecture.

Nested loops and merge join Early work on interval joins [18,32] studied a temporal join problem, where two relations are equi-joined on a non-temporal attribute and the temporal overlaps of joined tuple pairs should also be identified. Techniques based on nested-loops (for unordered inputs) and on sort-merge join (for ordered inputs) were proposed, as well as specialized data structures for append-only databases. Similar to plane sweep, merge join algorithms require the two input collections to be sorted, but join computation is sub-optimal compared to FS, which guarantees at most $|R| + |S|$ comparisons that do not produce results.

Index-based algorithms Enderle et al. [15] propose interval join algorithms, which operate on two RI-trees [23] that index the input collections. Zhang et al. [37] focus on finding pairs of records in a temporal database that intersect in the (key, time) space (i.e., a problem similar to that studied in [18,32]), proposing an extension of the multi-version B-tree [3].

Partitioning-based algorithms A partitioning-based approach for interval joins was proposed in [34]. The domain is split into disjoint ranges. Each interval is assigned to the partition corresponding to the last domain range it overlaps. The domain ranges 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. Histogram-based techniques for defining good partition boundaries were proposed in [33]. A more sophisticated partitioning approach, called Overlap Interval Partitioning (OIP) Join [13], divides the domain into equal-sized granules and consecutive granules define the ranges of the partitions. Each interval is assigned to the partition corresponding to the smallest sequence of granules that contains it. In the join phase, partitions of one collection are joined with their overlapping partitions from the other collection. OIP was shown to be superior compared to index-based approaches [15] and sort-merge join. These results are consistent with the comparative study of [16], which shows that partitioning-based methods are superior to nested loops and merge join approaches.

Disjoint Interval Partitioning (DIP) [8] was recently proposed for temporal joins and other sort-based operations on interval data (e.g, temporal aggregation). The main idea behind DIP is to divide each of the two input relations into partitions, such that each partition contains only disjoint intervals. Every partition of one input is then joined with all of the other. Since intervals in the same partition do not overlap, sort-merge computations are performed without backtracking. Prior to this work, temporal aggregation was studied in [26]. Given a large collection of intervals (possibly associated with values), the objective is to compute an aggregate (e.g., count the valid intervals) at all points in time. An algorithm was proposed in [26] which divides the domain into partitions (buckets), assigns the intervals to the first and last bucket they overlap and maintains a meta-array structure for the aggregates of buckets entirely covered by intervals. The aggregation can then be processed independently for each bucket (e.g., using a sort-merge based approach) and the algorithm can be parallelized in a shared-nothing architecture. We also propose a domain-partitioning approach for parallel processing (Sect. 7), but the details differ due to the different natures of temporal join and aggregation.

Methods based on plane sweep The Endpoint-Based Interval (EBI) Join [29] (reviewed in Sect. 3.1) and its lazy version LEBI were shown to significantly outperform OIP [13] and to also be superior to another plane sweep implementation [2]. An approach similar to EBI is used in SAP HANA [21]. To our knowledge, no previous work was compared to FS [7] (detailed in Sect. 3.2). In Sect. 4, we propose four optimizations for FS that greatly improve its performance, making it competitive or even faster than LEBI. Last, extensions and applications of the plane sweep approach has been discussed in [6,10], but in the context of temporal aggregation and SPARQL query processing, respectively.

Parallel algorithms A domain-based partitioning strategy for interval joins on multi-processor machines was proposed in [24]. Each partition is assigned to a processor and intervals are replicated to the partitions they overlap, to allow join results being produced independently at each processor. At the end, a merge phase with duplicate elimination is required as the same join result can be produced by different processors. Duplicates can be avoided using the reference test from [14] but, this approach incurs extra comparisons. Our parallel processing approach in Sect. 7 also applies a domain-based partitioning but produces no duplicates. Also, we propose a

ALGORITHM 1: Endpoint-Based Interval Join (EBI)

Input	: collections of intervals R and S
Output	: all intersecting pairs $(r, s) \in R \times S$
Variables	: endpoint indices EI^R and EI^S, active interval sets A^R and A^S

```
1  A^R ← ∅, A^S ← ∅;
2  build EI^R and EI^S;
3  sort EI^R and EI^S first by endpoint then by type;
4  e^R ← first index tuple in EI^R;
5  e^S ← first index tuple in EI^S;
6  while EI^R and EI^S not depleted do
7      if e^R < e^S then
8          if e^R.type = START then
9              r ← interval in R with identifier e^R.id;
10             add r to A^R;                       ▷ r is open
11             foreach s ∈ A^S do
12                 output (r, s);                  ▷ update result
13         else
14             remove r from A^R;                  ▷ r no longer open
15         e^R ← next index tuple in EI^R;
16     else
17         if e^S.type = START then
18             s ← interval in S with identifier e^S.id;
19             add s to A^S;                       ▷ s is open
20             foreach r ∈ A^R do
21                 output (r, s);                  ▷ update result
22         else
23             remove s from A^S;                  ▷ s no longer open
24         e^S ← next index tuple in EI^S;
```

breakdown of each partition join to a set of mini-join jobs, which has never been considered in previous work.

Distributed algorithms Distributed interval joins were studied in [22]. The goal is to join sets of intervals located at different clients. The clients iteratively exchange statistics with the server, which help the latter to compute a coarse-level approximate join; exact results are refined by on-demand communication with the clients. Chawda et al. [9] implement the partitioning algorithm of [24] in the MapReduce framework and extend it to operate for other (non-overlap) join predicates. The main goal of distributed algorithms is to minimize the communication cost between the machines that hold the data and compute the join.

3 Plane sweep for interval joins

This section presents the necessary background on plane sweep based computation of interval joins. First, we detail the EBI algorithm [29]. Then, we review the forward scan based algorithm from [7], which has been overlooked by previous work. Both methods take as input collections R, S of intervals and compute all (r, s) pairs with $r \in R, s \in S$, that intersect. We denote by r.start (r.end) the starting (ending) endpoint of an interval r.

3.1 Endpoint-Based Interval Join

EBI [29] is based on the internal-memory plane sweep technique of [31], but tailored to modern hardware. Algorithm 1 illustrates the pseudo-code of EBI. EBI represents each input interval, e.g., $r \in R$, by two tuples in the form of ⟨endpoint, type, id⟩, where endpoint equals either r.start or r.end, type flags whether endpoint is a starting or an ending endpoint, and id is the identifier of r. These tuples are stored inside the *endpoint indices* EI^R and EI^S, sorted primarily by their endpoint and secondarily by type. To compute the join, EBI concurrently scans the endpoint indices, accessing their tuples in increasing global order of their sorting key, simulating a "sweep line" that stops at each endpoint from either R or S. At each position of the sweep line, EBI keeps track of the intervals that have started but not finished, i.e., the index tuples that are start endpoints, for which the index tuple having the corresponding end endpoint has not been accessed yet. Such intervals are called *active* and they are stored inside sets A^R and A^S; EBI updates these active sets depending on the type entry of current index tuple (Lines 10 and 14 for collection R and Lines 19 and 23 for S). Finally, for a current index tuple (e.g., e^R) of type *START*, the algorithm iterates through the active intervals of the opposite input (e.g., A^S on Lines 11–12) to produce the next bunch of results (e.g., the intervals of S that join with e^R.id).

By recording the active intervals from each collection, EBI can directly report the join results without any endpoint comparisons. To achieve this, the algorithm needs to store and scan the endpoint indices which contain twice the amount of entries compared to the input collections. Hence excluding the sorting cost for EI^R and EI^S, EBI conducts $2 \cdot (|R| + |S|)$ endpoint comparisons to advance the sweep line, in total. However, the critical overhead of EBI is the maintenance and scanning of the active sets at each loop; i.e., Lines 10 and 19 (add), Lines 11–12 and 20–21 (scan), Lines 14 and 23 (remove). This overhead can be quite high; for example, typical hash map data structures support efficient $O(1)$ updates but scanning their contents is slow. To deal with this issue, Piatov et al. designed a special hash table termed the *gapless hash map* which efficiently supports all three insert, remove and getNext operations. Finally, the authors further optimized the join computation by proposing a *lazy evaluation* technique which buffers consecutive index tuples of type *START* (and hence, their corresponding intervals) as long as they originate from the same input (e.g., R). When producing the join results, a *single* scan over the active set of the opposite collection (e.g., A^S) is performed for the entire buffer. This idea is captured by the *Lazy Endpoint-Based Interval* (LEBI) Join algorithm. By keeping the buffer size small enough to fit inside the L1 cache or even the cache registers, LEBI greatly reduces

ALGORITHM 2: Forward Scan based Plane Sweep (FS)

Input : collections of intervals R and S
Output : all intersecting pairs $(r, s) \in R \times S$

```
1  sort R and S by start endpoint;
2  r ← first interval in R;
3  s ← first interval in S;
4  while R and S not depleted do
5      if r.start < s.start then
6          s' ← s;
7          while s' ≠ null and r.end ≥ s'.start 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'.start do
14             output (r', s);                    ▷ update result
15             r' ← next interval in R;           ▷ scan forward
16         s ← next interval in S;
```

main memory cache misses and hence, outperforms EBI even more.

3.2 Forward scan-based plane sweep

The experiments in [29] showed that LEBI outperforms not only EBI, but also the plane sweep algorithm of [2], which directly scans the inputs ordered by start endpoint and keeps track of the active intervals in a linked list. Intuitively, both approaches perform a *backward scan*, i.e., a scan of already encountered intervals, organized by a data structure that supports scans and updates. In practice however, the need to implement a special structure may limit the applicability and the adoption of these evaluation approaches while also increasing the memory space requirements.

In [7], Brinkhoff et al. presented a different implementation of plane sweep, which performs a *forward scan* directly on the input collections and hence, (i) there is no need to keep track of active sets in a special data structure and (ii) data scans are conducted sequentially. [2] Algorithm 2 illustrates the pseudo-code of this method, denoted by FS. First, both inputs are sorted by the start endpoint of each interval. Then, FS *sweeps* a line, which stops at the start endpoint of all intervals of R, S in order. For each position of the sweep line, corresponding to the start of an interval, say $r \in R$, the algorithm produces join results by combining r with all intervals from the opposite collection, that start (i) after the sweep line and (ii) before r.end, i.e., all $s' \in S$ with r.start $\leq s'$.start $\leq r$.end (internal while-loops on Lines 7–10 and 13–16). Excluding the cost of sorting R and S, FS conducts $|R| + |S| + |R \bowtie S|$ point comparisons, in total. Specifically, each interval $r \in R$ (the case for S is symmet-

---

[2] The algorithm originally targets intersection join of 2D rectangles, but it is straightforward to apply for interval joins.

ric) is compared to just one $s' \in S$ which does not intersect r in the loop at Lines 8–10.

4 Optimizing FS

We present four optimization techniques for FS that can greatly enhance its performance. Naturally, the cost of FS cannot be asymptotically reduced; $|R| + |S|$ endpoint comparisons is the unavoidable cost of advancing the sweep line. However, it is possible to reduce the number of $|R \bowtie S|$ comparisons required to produce the join results, which is the focus of the first two optimization techniques termed *grouping* and *bucket indexing*. In addition, low level code engineering and careful data layout in main memory can further improve the running time of FS, which is the focus of our *enhanced loop unrolling* and *decomposed data layout* techniques.

4.1 Grouping

The intuition behind our first optimization technique is to group consecutively swept intervals from the same collection and produce join results for them in batch, avoiding redundant comparisons. We exemplify this idea in Fig. 2, which depicts intervals $\{r_1, r_2\} \in R$ and $\{s_1, s_2, s_3, s_4, s_5\} \in S$ sorted by start endpoint. Assume that FS has already examined s_1; since r_1.start $< s_2$.start, the next interval where the sweep line stops is r_1. Algorithm 2 (Lines 7–10) then forwardly scans through the shaded area in Fig. 2a from s_2.start until it reaches s_5.start $> r_1$.end, producing result pairs $\{(r_1, s_2), (r_1, s_3), (r_1, s_4)\}$. The next stop of the sweep line is r_2.start, since r_2.start $< s_2$.start. FS scans through the shaded area in Fig. 2b producing results $\{(r_2, s_2), (r_2, s_3)\}$. We observe that the scanned areas of r_1 and r_2 are not disjoint, which in practice means that FS performed redundant endpoint comparisons. Indeed, this is the case for s_2.start and s_3.start which were compared to both r_1.end and r_2.end. However, since r_1.end $> r_2$.end holds, r_2.end $> s_2$.start automatically implies that r_1.end $> s_2$.start; therefore, pairs $(r_1, s_2), (r_2, s_2)$ could have been reported by comparing only r_2.end to s_2.start. Hence, processing consecutively swept intervals from the same collection (e.g., r_1 and r_2) *as a group* allows us to scan their common areas only once.

Algorithm 3 illustrates the pseudo-code of gFS, which enhances FS with the *grouping optimization*. Instead of processing a *single* interval at a time, gFS considers a *group* of consecutive intervals from the same collection at a time. Specifically, assume that at the current loop r.start $< s$.start (the other case is symmetric). Starting from r, gFS accesses all $r' \in R$ with r'.start $< s$.start (Line 7) and puts them in a group G^R. Next, the contents of G^R are *reordered* by increasing end endpoint (Line 8). Then, gFS initiates a *for-*
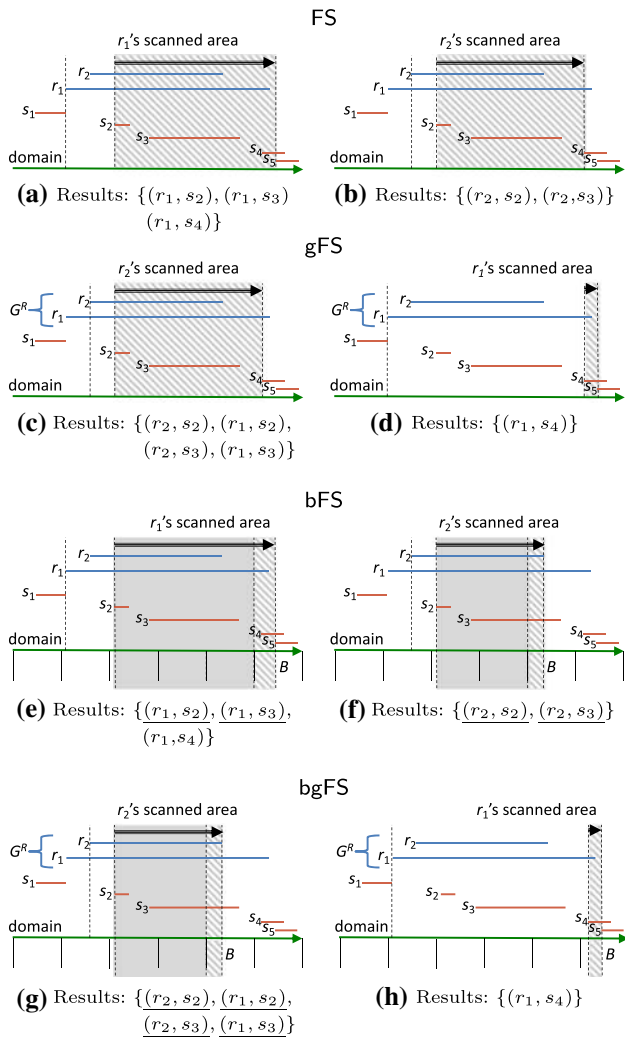
Fig. 2 Scanned areas by FS, gFS, bFS and bgFS for r_1 and r_2; with grouping r_2 precedes r_1. Underlined result pairs are produced without any endpoint comparisons

ward scan on S starting from $s' = s$ (Lines 9–14), but unlike FS the scan is done only once for all intervals in G^R. For each $r_i \in G^R$ in the new order, if s'.start $\leq r_i$.end, then s' intersects not only r_i but also all intervals in G^R *after* r_i (due to the sorting of G^R by end). If s'.start $> r_i$.end, then s' does not join with r_i but may join with succeeding intervals in G^R, so the for loop proceeds to the next $r_i \in G^R$.

Figure 2c, d exemplify gFS for intervals r_1 and r_2 grouped under G^R; as r_1.end $> r_2$.end, r_2 is considered first. When the shaded area in Fig. 2c from s_2.start until s_4.start is scanned, gFS produces results that pair both r_2 and r_1 with covered intervals s_2 and s_3 from S, by comparing s_2.start and s_3.start only to r_2.end. Intuitively, avoiding redundant endpoint comparisons corresponds to removing the overlap between the scanned areas of consecutive intervals; compare r_1's scanned area by gFS in Fig. 2d to the area in Fig. 2b by FS after removing the overlap with r_2's area.

ALGORITHM 3: FS with grouping (gFS)

Input : collections of intervals R and S
Output : all intersecting pairs $(r, s) \in R \times S$
Variables : groups G^R and G^S

1 **sort** R and S by start endpoint;
2 $r \leftarrow$ first interval in R;
3 $s \leftarrow$ first interval in S;
4 **while** *R and S not depleted* **do**
5   **if** r.start $< s$.start **then**
6        $G^R \leftarrow$ next group from R w.r.t. r, s;
7   **sort** G^R by end endpoint;
8        $s' \leftarrow s$;
9   **foreach** $r_i \in G^R$ *in order* **do**
10   **while** $s' \neq null$ **and** s'.start $\leq r_i$.end **do**
11   **foreach** $r_j \in G^R, j \geq i$ **do**
12   **output** (r_j, s');   ▷ update result
13              $s' \leftarrow$ next interval in S;   ▷ scan forward
14        $r \leftarrow$ first interval in R after G^R;
15   **else**
16        $G^S \leftarrow$ next group from S w.r.t. s, r;
17   **sort** G^S by end endpoint;
18        $r' \leftarrow r$;
19   **foreach** $s_i \in G^S$ *in order* **do**
20   **while** $r' \neq null$ **and** r'.start $\leq s_i$.end **do**
21   **foreach** $s_j \in G^S, j \geq i$ **do**
22   **output** (r', s_j);   ▷ update result
23              $r' \leftarrow$ next interval in R;   ▷ scan forward
24        $s \leftarrow$ first interval in S after G^S;

Discussion and implementation details The grouping technique of gFS differs from the buffering employed by LEBI [29]. First, LEBI groups consecutive start endpoints in a sort order that includes 4 sets of items, whereas in gFS there are only 2 sets of items (i.e., only start endpoints of the two collections). As a result, the groups in gFS are likely to be larger than LEBI's buffer (and larger groups make gFS more efficient). Second, the buffer in LEBI is solely employed for outputting results while groups in gFS also facilitate the avoidance of redundant endpoint comparisons due to the reordering of groups by end endpoint.

Regarding the implementation of grouping in gFS, we experimented with two different approaches. In the first approach, each group is copied to and managed in a dedicated array in main memory. The second approach retains pointers to the begin and end index of each group in the corresponding collection; the segment of the collection corresponding to the group is re-sorted (note that correctness is not affected by this). Our tests showed that the first approach is always faster, due to the reduction of cache misses during the multiple scans of the group (i.e., Lines 12-13 and Lines 22-23).

4.2 Bucket indexing

Our second optimization technique extends FS to avoid even more endpoint comparisons during the computation of the join results. The idea is as follows. First, we split the domain
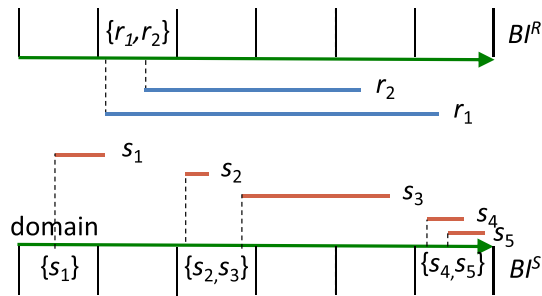
Fig. 3 Bucket indexing: domain stripes and BI^R, BI^S bucket indices for the intervals of Fig. 2

into a predefined number of equally-sized disjoint stripes; all intervals from R (resp. S) that start within a particular stripe are stored inside a dedicated bucket of the BI^R (resp. BI^S) *bucket index*. Figure 3 exemplifies the domain stripes and the bucket indices for the interval collections of Fig. 2.[3]

ALGORITHM 4: FS with bucket indexing (bFS)

Input	: collections of intervals R and S
Output	: all intersecting pairs $(r, s) \in R \times S$
Variables	: bucket indices BI^R and BI^S

1   **sort** R and S by start endpoint;
2   **build** BI^R and BI^S;
3   $r \leftarrow$ first interval in R;
4   $s \leftarrow$ first interval in S;
5   **while** *R and S not depleted* **do**
6   **if** $r.\text{start} < s.\text{start}$ **then**
7       $s' \leftarrow s$;
8       $B \leftarrow$ bucket in BI^S: $B.\text{start} \leq r.\text{end} < B.\text{end}$;
9   **while** *s' is before B* **do**   ▷ no comparisons
10   **output** (r, s');   ▷ update result
11         $s' \leftarrow$ next interval in S;   ▷ scan forward
12   **while** $s' \neq null$ **and** $s'.\text{start} \leq r.\text{end}$ **do**
13   **output** (r, s');   ▷ update result
14         $s' \leftarrow$ next interval in S;   ▷ scan forward
15       $r \leftarrow$ next interval in R;
16   **else**
17       $r' \leftarrow r$;
18       $B \leftarrow$ bucket in BI^R: $B.\text{start} \leq s_i.\text{end} < B.\text{end}$;
19   **while** *r' is before B* **do**   ▷ no comparisons
20   **output** (r', s);   ▷ update result
21         $r' \leftarrow$ next interval in R;   ▷ scan forward
22   **while** $r' \neq null$ **and** $s.\text{end} \geq r'.\text{start}$ **do**
23   **output** (r', s);   ▷ update result
24         $r' \leftarrow$ next interval in R;   ▷ scan forward
25       $s \leftarrow$ next interval in S;

With the bucket indices, the area scanned by FS for an interval is entirely covered by a range of stripes. Consider Fig. 2c, e; r_1's scanned area lies inside four stripes which means that the involved intervals from S start between the

---

[3] A bucket may in fact be empty; however, we can control the ratio of empty buckets by properly setting the total number of stripes while in practice, empty buckets mostly occur for very skewed distributions of the start endpoints.

BI^S bucket covering $s_2.\text{start}$ and the BI^S bucket covering $r_1.\text{end}$. In this spirit, area scanning resembles a range query over the bucket indices. Hence, every interval s_i from a bucket completely inside r_1's scanned area or lying after s_2 in the first bucket, can be paired to r_1 as join result without any endpoint comparisons; by definition of the stripes/buckets, for such intervals $s_i.\text{start} \leq r_1.\text{end}$. So, we only need to conduct endpoint comparisons for the s_i intervals from the bucket that covers $r_1.\text{end}$. This distinction is graphically shown in Fig. 2e, f where solid gray areas are used to directly produce join results with no endpoint comparisons. Observe that, for this example, both join results produced when FS performs a forward scan for r_2 are directly reported when using the bucket indexing. On the other hand, bucket indexing enables us to directly report only two of the three join results for r_1 as the bucket that contains s_4 is not completely inside r_1's scanned area.

Algorithm 4 illustrates the pseudo-code of bFS which enhances FS with *bucket indexing*. Essentially, bFS operates similar to FS. Their main difference lies in the forward scan for the current interval. Without loss of generality, consider $r \in R$ (the case of $s \in S$ is symmetric); Lines 8–14 implement the range query discussed in the previous paragraph. The algorithm first identifies bucket $B \in BI^S$ which covers $r.\text{end}$. Then, it iterates through the $s' \in S$ intervals after current s, originating from all buckets before B to directly produce join results on Lines 9–11 without any endpoint comparisons, while finally on Lines 12–14, the intervals of B are scanned and compared exactly as in FS.

Discussion and implementation details In our implementation, we choose not to materialize the index buckets, i.e., no intervals are copied to dedicated data structures. We store for each bucket a pointer to the last interval in it; this allows bFS to efficiently perform the forward scans. With this design, we guarantee a small main memory footprint for our method as there is no need to practically store a second copy of the data.

4.3 Enhanced loop unrolling

Our third optimization builds upon a code transformation technique known as *loop unrolling* or *loop unwinding* [1,27, 28]. Essentially, the goal of loop unrolling is to reduce the execution time by (i) eliminating the overhead of controlling a loop (i.e., checking its exit condition) and the latency due to main memory accesses, and (ii) reducing branch penalties. Such a transformation can be carried either manually by the programmer or automatically by the compiler; our focus is on the former case.

The idea of manual unrolling involves the re-writing of the loop as a repeated sequence of similar independent statements. For example, a loop which processes the 1000 elements of an array can be modified to perform only 100 iterations using a so-called unrolling *factor* of 10; i.e.,
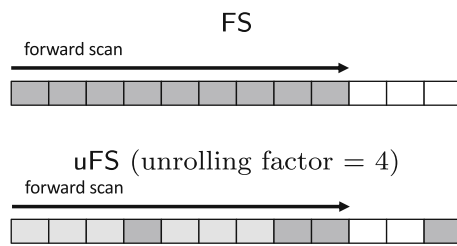
Fig. 4 Enhanced loop unrolling: forward scans. Endpoint comparisons are colored in dark gray; direct results output with no comparisons are in light gray

Fig. 5 Decomposed data layout: sweeping and scans

every iteration of the new loop executes 10 *identical* and *independent* element processing statements. In this spirit, a straightforward way to benefit from loop unrolling will be to unfold the forward scan loop on Lines 7–9 of Algorithm 2 (the case of Lines 13–15 is symmetric) by a factor of x. Under this, the exit condition $s' \neq null$ will checked only once for every x-th interval. Also, every iteration of the new loop checks the $r.\text{end} \geq s'.\text{start}$ overlap condition on each of the next x (r, s') pairs and if so, the pair is output.

Despite its positive effect on reducing the loop cost, this straightforward approach would still incur the same number of endpoint comparisons as the forward scan of FS, because the $r.\text{end} \geq s'.\text{start}$ condition is checked for every reported pair. In view of this, we propose an adaptation termed the *enhanced loop unrolling* which *skips* endpoint comparisons to accelerate FS. Specifically, instead of checking $r.\text{end} \geq s'.\text{start}$ for every (r, s') pair, we check whether this condition holds for the x-th s'. If so, all x intervals are guaranteed to pair with current interval r, the x pairs are reported without the need of any comparisons, and we proceed to the next x intervals. Otherwise (i.e., if $r.\text{end} < s'.\text{start}$) the x-th s' interval does not overlap r and therefore, we need to scan the $x - 1$ intervals similar to FS. We denote by uFS the extension of FS which employs the enhanced loop unrolling optimization.

Figure 4 illustrates the functionality and the effect of the enhanced loop unrolling. Fix current interval r from collection R, which overlaps with 8 intervals from S. The forward scan of FS accesses 9 s' intervals, conducting 9 endpoint comparisons for the $r.\text{end} \geq s'.\text{start}$ condition. The last comparison is needed to terminate the forward scan, i.e., it determines the first s' interval that starts after $r.\text{end}$. On the other hand, uFS with an unrolling factor of 4 requires only 4 endpoint comparisons, in total. Specifically, the $r.\text{end} \geq s'.\text{start}$ condition is initially checked for the fourth interval in S; since, the condition holds, all first 4 s' intervals overlap current r. The next 4 s' intervals are examined in the same manner. Last, uFS checks the $r.\text{end} \geq s'.\text{start}$ condition for the twelfth s' interval. As $r.\text{end} < s'.\text{start}$, the twelfth interval from S does not overlap r, which means that uFS will complete the forwards scan similar to FS conducting an extra endpoint comparison for the ninth interval.
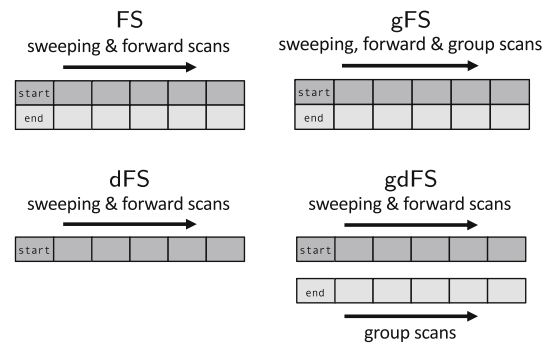
4.4 Decomposed data layout

We can further enhance FS by carefully storing the input interval, in main memory. To demonstrate our intuition, consider again Algorithm 2 and the pseudo-code of FS. The algorithm essentially performs two operations; it advances the sweep line and forwardly scans the collections. We observe that neither of these operations considers every attribute from the input intervals. Specifically, in order to advance the sweep line the start endpoints of the current intervals $r \in R$ and $s \in S$ are compared, while end endpoints are of no use. Concerning forward scanning, assume without loss of generality that the current (fixed) interval is $r \in R$ and so, FS will next scan collection S (the case of forwardly scanning R is symmetric). Essentially, the algorithm needs only the end endpoint of *current* interval r and the start endpoint of *every* scanned interval s' from S, in order to check the $r.\text{end} \geq s'.\text{start}$ condition in Line 7. On the other hand, both $r.\text{start}$ and $s'.\text{end}$ for every examined s' are of no use to the forward scan operation.

Based on this observation, our last technique is inspired by the *Decomposition Storage Model* (DSM) [12], adopted by column-oriented database systems (e.g., [35]). Instead of storing an input collection as an array of ⟨start, end⟩ tuples, we decompose it into two separate arrays; one having the start endpoints and one with end endpoints. With this decomposition, the algorithm can iterate only over the start arrays when advancing the sweep line or forward scanning, which results in a smaller footprint in main memory and reduces the number of cache misses. We denote by dFS the extension of FS that employs our decomposed data layout. Figure 5 illustrates our decomposed data layout for dFS compared to the data layout for FS.

4.5 Employing all optimizations

We finally discuss how all proposed optimization techniques can be put together in FS. Grouping and bucket indexing optimize FS in an orthogonal manner; hence, it is possible

to pair the optimizations resulting to what we call FS with grouping and bucket indexing (bgFS). Figures 2(g) and (h) exemplify bgFS for intervals r_2 and r_1 (sorted by end endpoints) and their group G^R. Compared to bFS, the algorithm iterates through the same buckets regarding r_2's scanned area, but produces join results for both r_2 and r_1 at the same time, similarly to gFS. Regarding r_1's scanned area, bgFS operates exactly as gFS since the area is covered by a single bucket.

Essentially, the pseudo-code of bgFS would resemble Algorithm 4 of bFS with the exceptions of having to form groups and how the forward scans are performed. Similar to gFS and Lines 6–7 in Algorithm 3, bgFS groups together consecutive intervals from the same input and reorders the contents of each group by their increasing end endpoint. Then, Lines 9–11 and 12–14 are adjusted according to Lines 9–13 in Algorithm 3 of gFS, where a forward scan is performed for an entire group instead of a single interval. The case of grouping on collection S is symmetric.

The performance of bgFS can be further improved by the enhanced loop unrolling and adopting the decomposed data layout. Plugging enhanced loop unrolling into bgFS is straightforward and so is pairing our decomposed data layout with bucket indexing. Grouping can be enhanced by carefully decomposing the group data. Without loss of generality consider gFS; the same approach can be applied for bgFS and bguFS. Similarly to FS, we observe that forward scans on collection S in (Lines 9–13, Algorithm 3) take into account only the end endpoint of each interval in group G^R (the case of forward scanning R is symmetric). In fact, start for r intervals is used only to form the group in Line 6 before the forward scan commences. Hence, we can model every group as two arrays. Figure 5 illustrates this idea. Originally, all gFS operations are conducted under the original layout where both the input collections and created groups are stored in arrays of ⟨start, end⟩ tuples. In contrast, by employing our decomposed layout advancing the sweep line and forward scan operations use only the start arrays whereas group scans (i.e., the for loops in Line 9 and 19) operate on the end arrays.

In Sect. 6.2, we experimentally study the effect of each of the four proposed optimization techniques. We also provide insights on how we can decide which of them should be activated depending on the characteristics of the input collections. To this end, we devise the optFS method in Sect. 6.3.

5 The case of self-joins

Up to this point, we investigated only the case where the intervals from two distinct collections are joined. In this section, we discuss the case of a *self-join*, which receives a single collection as input R and looks for the pairs of intervals $(r_i, r_j) \subseteq R \times R$ that overlap. All interval join

---

ALGORITHM 5: Self-join FS

Input   : collection of intervals R
Output   : all intersecting pairs $(r, r') \in R \times R$

1 **sort** R by start endpoint;
2 $r \leftarrow$ first interval in R;
3 **while** R *not depleted* **do**
4   **output** (r, r);   ▷ update result
5      $r' \leftarrow$ interval right after r in R;
6   **while** $r' \neq null$ **and** $r.end \geq r'.start$ **do**
7   **output** (r, r');   ▷ update result
8          $r' \leftarrow$ next interval in R;   ▷ scan forward
9      $r \leftarrow$ next interval in R;

---

algorithms, which we have discussed already, can be directly applied to solve this problem, if we set the second input $S = R$. However, such an approach requires a *duplicate elimination* post-processing step (or an extra comparison for each computed pair), otherwise every (r_i, r_j) would be reported twice, increasing the total number of results to $(2 \cdot |R \bowtie R| - |R|)$. Consider, for example, the collection $R = \{r_1[3, 5], r_2[4, 6], r_3[7, 11]\}$. The result of the $R \bowtie R$ self-join contains pairs (r_1, r_1), (r_1, r_2), (r_2, r_2) and (r_3, r_3). Now, assume we use FS from Algorithm 2 to compute this join by setting $S = R$. The sweep line will first stop at r_1; the forward scan on S will start from s_1 and output (r_1, s_1) and (r_1, s_2), which correspond to (r_1, r_1) and (r_1, r_2). The next interval will be s_1; the forward scan will start from the current interval from R, which was set to r_2 at the end of the first forward scan, and hence, output (s_1, r_2) (i.e., (r_1, r_2)) for a second time.

To address this issue, we design a simplified version of FS which pairs an interval r only with itself and intervals from the collection that come after r in the sort order.[4] Algorithm 5 illustrates the pseudo-code for the self-join version of FS. Going back to the previous example, the forward scan for r_1 will produce (r_1, r_2) but the forward scan for r_2 will start from r_3 and so, avoid duplicate results.

All our proposed optimizations can be applied on the self-join FS. The case of bucket indexing is straightforward; in practice, only one bucket index is defined and Algorithm 5 is extended accordingly to Algorithm 4. Enhanced loop unrolling and decomposed data layout for self-joins operate exactly as discussed in Sects. 4.3 and 4.4, respectively. On the other hand, we reconsider our grouping optimization, as all intervals are essentially consecutive from the same input. The solution is to group together intervals with exactly the same start endpoint. Last, special care is taken for the group scan of gFS (i.e., corresponding to the for loop in Lines 9 and 9, Algorithm 3). Specifically, to avoid duplicate results the i-th interval of a group G is paired to itself and the $|G| - i$

---

[4] A similar approach can be taken for EBI/LEBI; in this case, we maintain only one active set A.

Table 1 Characteristics of experimental datasets

	BOOKS [5]	FLIGHTS [6]	GREEND [8,25]	INFECTIOUS [8,19]	TAXIS	WEBKIT [5,6,13]
Cardinality	2,312,602	445,827	110,115,441	415,912	172,668,003	2,347,346
Domain duration (secs)	31,507,200	2,750,280	283,356,410	6,946,360	31,768,287	461, 829, 284
Distinct endpoints	5330	41,975	182,028,123	81,514	29,873,023	174,471
Shortest interval (secs)	1	1260	1	20	1	1
Avg. interval duration (secs)	2,201,320	8790	15	20	758	33, 206, 300
Longest interval (secs)	31,406,400	42,300	59,468,008	20	2,148,385	461,815,512

intervals that come after it inside G, in the sort order. Note that these results can be reported while constructing the group.

6 Experiments on single-threaded processing

We next present the first part of our experimental analysis on the single-threaded computation of interval joins.

6.1 Setup

Our single-threaded analysis was conducted on a machine with 384 GBs of RAM and a dual Intel(R) Xeon(R) CPU E5-2630 v4 clocked at 2.20GHz running CentOS Linux 7.3.1611. All methods were implemented in C++, compiled using `gcc` (v4.8.5) with flags `-O3`, `-mavx` and `-march=native`. We imported in our source code the implementations of EBI/LEBI [29], OIP [13] and DIP [8], kindly provided by the authors of the corresponding papers. The setup of our benchmark is similar to [29]; every interval contains two 64-bit endpoint attributes (i.e., start and end) while the workload accumulates the sum of an XOR between the start attributes on every result pair. Note that all data (input collections, index structures etc.) reside in main memory.

Datasets We experimented with 6 real datasets, the majority of which was used in recent literature on interval joins; Table 1 details the characteristics of the datasets. BOOKS [5] records all transactions at Aarhus public libraries in 2013 (https://www.odaa.dk); valid times indicate the periods when a book is lent out. FLIGHTS [6] records domestic flights in USA during January 2016 (https://www.bts.gov); valid times indicate the duration of a flight. GREEND [8,25] records power usage data from households in Austria and Italy from January 2010 to October 2014; valid times indicate the period of a measurement. INFECTIOUS [8,19] stores visiting information from the "INFECTIOUS: stay Away!" exhibition at Science Gallery in Dublin, Ireland, from May to July 2009; valid times indicate when a contact between visitors occurred. TAXIS records taxi trips (pick-up, drop-
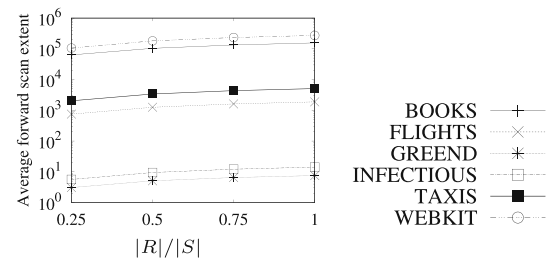
Fig. 6 Selectivity of the tested join queries

off timestamp) from New York City (https://www1.nyc.gov/site/tlc/index.page) in 2013; valid times indicate the duration of each ride. WEBKIT [5,6,13] records the file history in the git repository of the Webkit project from 2001 to 2016 (https://webkit.org); valid times indicate the periods when a file did not change.

Queries We ran a series of interval join queries using uniformly sampled subsets of each dataset as the outer input R and the entire dataset as the inner S; for each setting, the $|R|/|S|$ ratio varies inside $\{0.25, 0.5, 0.75, 1\}$.[5] To assess the performance of the evaluation methods, we measured their total execution time which includes sorting, indexing and partitioning costs (wherever applicable).

Figure 6 reports on the selectivity of our tested join queries; for each dataset and $|R|/|S|$ value, the figure plots how many intervals overlap with an input interval, on average. Under this, our datasets can be essentially divided into 3 categories. Joins on GREEND and INFECTIOUS are highly selective as every interval overlaps with at most 10 others, on average. In contrast, the result sets on WEBKIT and BOOKS queries include over 10, 000 pairs for each input interval, on average. Queries on FLIGHTS and TAXIS lie in the middle, but they are significantly less selective than the GREEND and INFECTIOUS joins.

Tuning To tune our bucket indexing optimization, we ran a test for the $|R| = |S|$ setting which monitored the execution time of bFS while varying the number B of buckets or equivalently the number of domain stripes used. Table 2 reports on the results of this test; the lowest execution time for each

---

[5] We also tested disjoint subsets observing similar behavior.

Table 2 Tuning bucket indexing: bFS execution time [secs] for $|R| = |S|$; lowest time in bold

# buckets B (or domain stripes)	BOOKS	FLIGHTS	GREEND	INFECTIOUS	TAXIS	WEBKIT
1	645	1.33	**9.74**	**0.022**	1, 464	1, 250
5	552	1.33	10.3	0.022	1, 345	1, 120
10	524	1.21	10.3	0.022	1, 340	1, 126
50	451	1.21	10.4	0.023	1, 332	1, 063
100	372	1.16	10.5	0.025	1, 314	914
500	355	0.92	10.4	0.026	1, 312	899
1000	353	0.72	10.5	0.025	1, 286	877
5000	348	0.56	10.1	0.024	1, 268	874
10,000	**347**	0.53	10.3	0.026	1, 281	872
50,000	350	**0.52**	10.9	0.027	1, 065	873
100,000	354	0.52	10.2	0.027	872	**865**
500,000	354	0.53	10.5	0.033	693	878
1,000,000	347	0.53	10.7	0.040	**645**	876
5,000,000	355	0.58	10.1	0.089	651	902
10,000,000	354	0.64	10.8	0.105	650	898

dataset is highlighted in bold. We draw two important findings. First, bucket indexing is not effective on GREEND and INFECTIOUS; the lowest execution time was observed for $B = 1$, i.e., when bFS operates exactly as FS. We elaborate on this issue in the next section. On the other hand, increasing the number of buckets accelerates bFS for BOOKS, FLIGHTS, TAXIS and WEBKIT joins. The best B value for all four datasets lies in between 10, 000 and 1, 000, 000; further increasing B eventually slows down bFS because the domain is fragmented in too many stripes. Under this, we set the number of buckets for the rest of this article to 100, 000. Last, we set the loop unrolling factor to 32, similar to previous work in [29], such that every loop iteration can be processed as high as possible in the main memory cache hierarchy.

6.2 Optimizing FS

We first study the effectiveness of our optimization techniques for FS, i.e., grouping, bucket indexing, enhanced loop unrolling and decomposed data layout, captured by methods gFS, bFS, uFS and dFS, respectively. Figure 7 reports the execution time of the methods. To save space, we do not include a breakdown for the execution time of the methods. Nevertheless, the findings are similar to the case of one partition in Figures 11 and 13, i.e., for highly selective queries, sorting dominates the total computation cost.

Grouping We observe that the grouping optimization is effective in 4 out of our 6 experimental datasets. In fact, the execution times in Fig. 7 align with the join selectivities in Fig. 6. For the highly selective queries in GREEND and INFECTIOUS, gFS is slower than FS. As these datasets contain very short intervals (see Table 1), a forward scan by FS
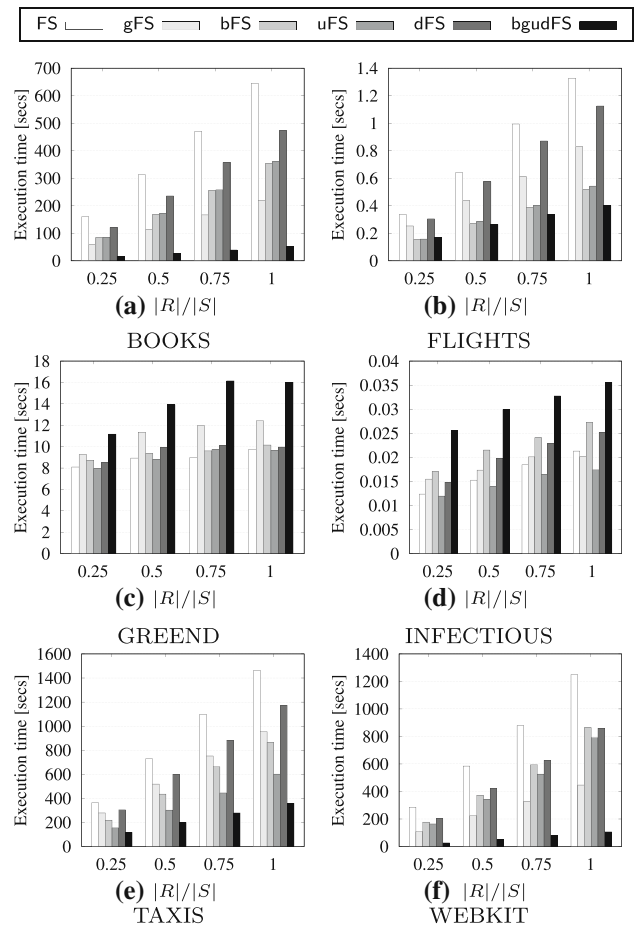

Fig. 7 Optimizing FS: execution time

Table 3 Grouping: extent of forward scan per input interval

Dataset	$|R|/|S|$							
	0.25		0.5		0.75		1	
	FS	gFS	FS	gFS	FS	gFS	FS	gFS
BOOKS	63,557	1149	103,769	1321	133,746	1406	156,001	1456
FLIGHTS	753	140	1257	185	1615	208	1885	220
GREEND	3.1	2.1	5.1	3.9	6.5	5.3	7.5	6.5
INFECTIOUS	5.6	0.5	9.5	0.8	12.2	1.2	14.2	1.5
TAXIS	2039	576	3398	893	4369	1069	5098	1169
WEBKIT	106,209	6943	181,776	11,029	233,422	13,713	272,945	15,408

Table 4 Grouping: average group size

Dataset	$|R|/|S|$			
	0.25	0.5	0.75	1
BOOKS	290	339	392	446
FLIGHTS	11.1	10.6	11.3	12.3
GREEND	2.7	1.7	1.4	1.2
INFECTIOUS	13.5	8.1	6.3	5.4
TAXIS	7.2	5.9	6	6.3
WEBKIT	14.4	12.2	12.3	13.4

Table 5 Bucket indexing: percentage of the join results produced without endpoint comparisons.

Dataset	$|R|/|S|$			
	0.25	0.5	0.75	1
BOOKS	77%	72%	75%	77%
FLIGHTS	60%	60%	60%	60%
GREEND	9%	9%	9%	9%
INFECTIOUS	0%	0%	0%	0%
TAXIS	49%	48%	48%	48%
WEBKIT	78%	73%	63%	59%

examines only a few intervals (10 or less on average, according to Fig. 6); recall that the forward scan for an interval, e.g., $r \in R$, extents from the first interval in S which starts after r.start until the first interval in S which starts after r.end. As a result, any reduction in the average extent of the forward scan achieved by gFS does not payoff in practice. Table 3 reports on the forward scan extent per interval by FS and gFS. [6] Grouping induces a clear *relative* reduction of this extent for INFECTIOUS (approximately, one order of magnitude), but in *absolute* numbers the forward scans were very short and thus, cheap in the first place. An additional indicator for the ineffectiveness of grouping is the size of the created groups, reported in Table 4. Notice that for GREEND queries, groups contain less than two intervals on average; hence, gFS does not provide any benefit over FS.

On the other hand, gFS significantly outperforms FS, by a wide margin (up to one order of magnitude), for BOOKS, WEBKIT, FLIGHTS and TAXIS where the join queries return a large number of results. As the intervals in these datasets are significantly longer compared to GREEND and INFECTIOUS, a forward scan by FS examines a large number of intervals and consequently conducts a large number of endpoint comparisons. In this context, grouping consecutive intervals from the same input and performing a single forward

scan for the entire group enables gFS to massively produce result pairs and avoid redundant comparisons. In fact, the performance gain of gFS over FS grows with $|R|/|S|$, as the extent of the forward scans increases and the join queries become computationally harder. Last, we observe that the effectiveness of grouping increases also with the size of the created groups; notice how much gFS outperforms FS in BOOKS where each group contains some hundreds of intervals.

Bucket indexing Similar to grouping, the effectiveness of the bucket indexing optimization depends on the extent of the forward scans. Recall from Sect. 4.2 that bFS performs the forward scans as range queries over the domain stripes; buckets for stripes entirely contained inside the forward scan areas provide direct join results, i.e., without the need for additional endpoint comparisons. The longer forward scans are, the more stripes are entirely covered and hence, a larger number of redundant comparisons are avoided. Under this, bFS outperforms FS for all $|R|/|S|$ values on BOOKS, FLIGHTS, TAXIS and WEBKIT queries, while FS is faster than bFS for GREEND and INFECTIOUS where forward scans are very short. Table 5 reports the ratio of the result pairs that bFS outputs without conducting any comparisons. For joins on GREEND and INFECTIOUS, bFS essentially operates similar to FS but with the extra cost of creating and querying the bucket indices. In contrast, for the rest of the datasets, bFS outputs from 48% to over 70% of the result pairs without any endpoint comparisons.

---

[6] Overall, gFS forwardly scans the same number of intervals as FS - otherwise, its result set would be incomplete. However, gFS manages to reduce the total number of conducted scans as it performs one scan *per group* instead of one scan *per interval*; this optimization is equivalent to reducing the extent of the forward scan *per* input interval.

Table 6 Enhanced loop unrolling: percentage of the join results produced without endpoint comparisons.

Dataset	$|R|/|S|$			
	0.25	0.5	0.75	1
BOOKS	97%	97%	97%	97%
FLIGHTS	97%	97%	97%	97%
GREEND	48%	59%	64%	68%
INFECTIOUS	75%	77%	79%	80%
TAXIS	97%	97%	97%	97%
WEBKIT	97%	97%	97%	97%

Enhanced loop unrolling Among all four proposed optimizations, the enhanced loop unrolling is the most robust. As Fig. 7 shows, the technique is very effective when forward scans are long, i.e., for all queries in BOOKS, FLIGHTS, TAXIS and WEBKIT, while for highly selective joins with short scans, i.e., in GREEND, INFECTIOUS, it is less effective but almost never slows down the computation. The ratio of the result pairs which uFS outputs without any endpoint comparisons supports this finding (see Table 6); note that even on the highly selective joins in GREEND and INFECTIOUS, uFS directly outputs 50% or more of the results.

Decomposed data layout Last, our decomposed data layout exhibits similar behavior to grouping and bucket indexing. Essentially, long forward scans incur a large main memory footprint and hence, scanning a smaller in bytes dedicated array for start endpoints can significantly reduce the cache misses. Under this, queries on BOOKS and WEBKIT benefit the most from applying dFS. In contrast, for GREEND and INFECTIOUS the extra cost of the decomposition does not payoff as data for the forward scans are already small enough to be handled in the highest levels of the cache.

Discussion Figure 7 also reports the execution time of bgudFS which employs all four optimizations at the same time. We observe that on BOOKS, FLIGHTS, TAXIS and WEBKIT queries, bgudFS clearly outperforms FS and all its variants that employ a single optimization; this is expected as the proposed techniques optimize FS in an orthogonal manner and so, can be effectively combined. Note that the performance gain of bgudFS over the rest of the methods actually grows with $|R|/|S|$. On the other hand, for GREEND and INFECTIOUS queries, the method inherits the shortcomings of grouping, bucket indexing and decomposed data layout which renders bgudFS the slowest method.

Our analysis on optimizing FS draws two key conclusions. First, the enhanced loop unrolling which builds upon code transformation should be always applied; uFS outperformed FS in almost all our test queries. Second, the less selective and hence, more computationally expensive an interval join is, the more effective grouping, bucket indexing and decomposed data layout will be. Under these observations, the most efficient FS variant is either bgudFS or uFS, depending on the selectivity of the interval join.

6.3 optFS: a self-tuning FS

To deal with this decision problem, we devised the optFS method which operates in two phases. In the first phase, optFS roughly estimates the average cost of a forward scan; we rely on sampling and executing uFS, for this purpose. In brief, we uniformly divide the domain into a predefined number of ranges (equal to 50) and let uFS run on a sample from both inputs (equal to $1^0/_{00}$), inside every range; practically, a simplified and very fast version of uFS, which only counts the extent of the conducted forward scans, is executed. This sampling-based process manages to approximate the real value for the average forward scan extent with a 18% relative error, on average. Although we could improve the accuracy by increasing the number of ranges we divide the domain and/or the sampling ratio, our goal is different. We are interested only in estimating the order of magnitude for the forward scans extent; in this context, the discussed sampling-based process achieves almost an 100% accuracy. Our tests has shown that 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; i.e., the case of GREEND and INFECTIOUS queries. Based on this observation, optFS decides whether to run uFS or bgudFS in its second phase. Note that the cost of the first (sampling and decision) phase of optFS is negligible compared to the cost of the second phase (joining); in our tests, sampling and decision making took only $3^0/_{00}$ of the total execution time by optFS, on average.

6.4 optFS against the competition

After optimizing FS, we compare our optFS against previous work, i.e., the partition-based methods DIP, OIP and the state-of-the-art plane sweep method LEBI. For the competitor methods, we enforced traditional loop unrolling whenever was possible. In addition, we included the bgFS method from our previous publication [5]. Figure 8 reports the execution times; as expected, the time of all methods rises while increasing the $|R|/|S|$ ratio. Observe however that the plane sweep based methods LEBI, bgFS-[5] and optFS always outperform their partition-based competitors, in most cases by orders of magnitude with the exception of GREEND queries where DIP performance is very close to LEBI. This finding fully aligns with the analysis in [29], where LEBI (and plane sweep based algorithms in general) was shown to outperform OIP.

For optFS against LEBI, the tests clearly show that we achieved our original goal. Optimized FS can be not only competitive to but also faster than state-of-the-art LEBI which,
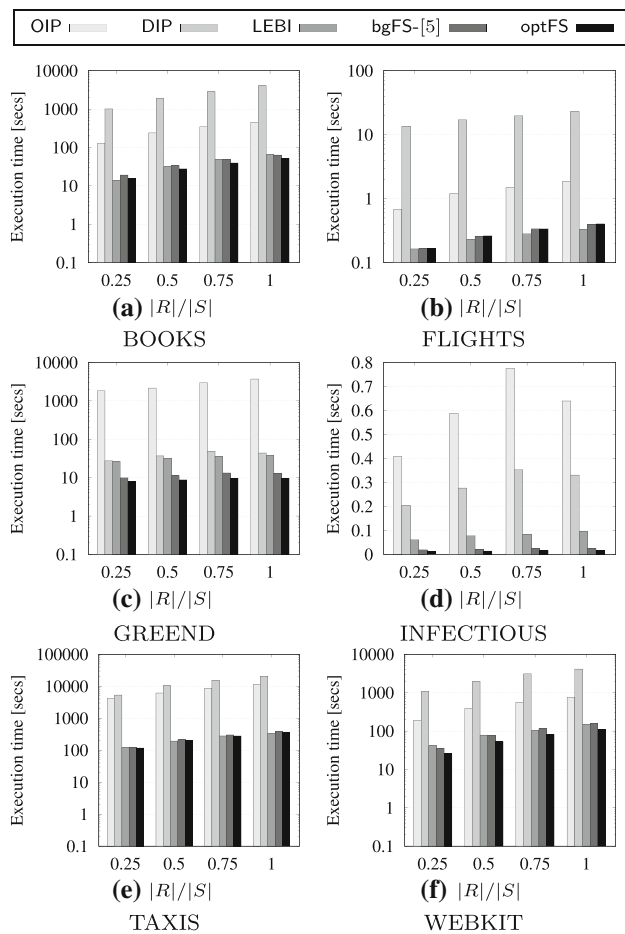
Fig. 8 Comparisons: optFS against competition

as discussed in Sect. 3.1, performs no endpoint comparisons to produce the results. Also, we made this possible without relying on a special data structure such as the gapless hash map. In fact, optFS outperforms LEBI in 16 of the 24 queries in Fig. 8. For the highly selective joins on GREEND and INFECTIOUS, optFS (powered by uFS) is faster by a 70-82% margin, while for the least selective joins on BOOKS and WEBKIT, optFS (powered by bgudFS) outperforms LEBI by a 13-36% margin. LEBI steadily outperforms optFS only on FLIGHTS by a 14-22% margin while on TAXIS the two methods have similar performance.

In terms of memory consumption, our preliminary analysis in [5] showed that LEBI always incurs a larger memory footprint than bgFS, due to the data replication from its endpoint indices and maintaining open intervals inside two gapless hash maps. The same trend holds compared to optFS. As a code transformation, enhanced loop unrolling incurs no extra storage costs, while the decomposed data layout results into a 19% average increase over bgFS, when used, i.e., for queries in BOOKS, FLIGHTS, TAXIS and WEBKIT.

In view of these results, our analysis in the rest of this article will primarily focus on optFS as the most efficient single-threaded method for interval joins.

7 Parallel processing

We now shift our focus to the parallel processing of interval joins that benefits from the existence of multiple CPU cores in a machine. We discuss three different solutions; (i) the case where no physical partitioning of the input collections is employed, (ii) the hash-based partitioning approach suggested in [29], and (iii) our domain-based partitioning approach. For the latter two approaches, we also discuss different strategies for efficiently partitioning the input intervals in Sect. 8.

7.1 No-partitioning parallel join

A straightforward approach to benefit from modern parallel hardware is to identify tasks of an interval join algorithm that are independent to each other and hence, can run in parallel. Every such task is assigned to a separate CPU core or thread. The input interval collections are never physically partitioned (hence, the name of the approach), which means that the processing threads need to simultaneously traverse data structures stored in shared main memory. A similar approach was used in the past for relational equi-joins, e.g., in [4], where a hash table is built in shared memory for the inner input and then, every thread reads a chunk of the outer and probes the shared hash table to produce join results.

Our experiments on single-threaded join computation clearly showed the advantage of plane sweep based evaluation and optFS in specific. In what follows, we discuss a no-partitioning parallel adaptation of FS and its variants. [7] Recall from Sect. 3.2 that the algorithm essentially involves two tasks; (i) advancing a sweep line which stops at the start endpoint of all input intervals, and (ii) for each position of the sweep line, performing a forward scan to output join results. Despite traversing the same data structures, i.e., those containing the input collections, it is easy to confirm that the forward scans are independent from each other. Therefore, we design a parallel version of FS which follows a master-slaves approach. We rely on a particular thread, which we call the *master*, to advance the sweep line, i.e., to execute Lines 4–5, 10–11 and 16 of Algorithm 2. When the sweep line stops, the master assigns the current forward scan to the next available thread (i.e., to a *slave*). Slave threads operate in a completely independent and asynchronous manner, executing instances of Lines 6–9 and 12–15 of Algorithm 2 in parallel. Note that all optimizations from Sect. 4 can be

---

[7] A similar approach can be employed for EBI/LEBI.

PARADIGM 1: Hash-based Partitioning

Input	: collections of intervals R and S, number of partitions k, hash function h
Output	: all intersecting interval pairs $(r, s) \in R \times S$

1 **foreach** *interval* $r \in R$ **do**   ▷ partition R
2     $v \leftarrow h(r)$;   ▷ apply hash function
3   **add** r to partition R_v;
4 **foreach** *interval* $s \in S$ **do**   ▷ partition S
5     $v \leftarrow h(s)$;   ▷ apply hash function
6   **add** s to partition S_v;
7 **foreach** *partition* R_i *of* R **do**
8   **foreach** *partition* S_j *of* S **do**
9   **compute** $R_i \bowtie S_j$;   ▷ LEBI, FS and variants

Fig. 9 Domain-based partitioning of the intervals in Fig. 2; the case of 4 domain stripes $t_1 \ldots t_4$

applied for parallel FS. Enhanced loop unrolling, decomposed data layout and bucket indexing are straightforward; for the latter, every slave thread will practically execute Lines 7–14 and 17–24 of Algorithm 4. For the grouping optimization, the master thread has to additionally create the groups (Lines 6 and 16 of Algorithm 3) but every group is then assigned to a slave thread which will first sort the group intervals according to their end endpoint and then perform the forward scan; in other words, a slave thread executes an instance of Lines 7–13 and 17–23 of Algorithm 3, receiving a group of intervals as input.

7.2 Hash-based partitioning

In [29], Piatov et al. proposed a *hash-based* partitioning paradigm for parallelizing EBI (and its lazy LEBI version), described by Paradigm 1. The evaluation of the join involves two phases. First, the input collections are split into k disjoint partitions using the same hash function h. During the second phase, a pairwise join is performed between all $\{R_1, \ldots, R_k\}$ partitions of collection R and all $\{S_1, \ldots, S_k\}$ of S; in practice, any single-threaded interval join algorithm can be employed to join two partitions. Since the partitions are disjoint, the pairwise joins run independently of each other.

In [29], the intervals in the input collections are sorted by their start endpoint before partitioning, and then assigned to partitions in a round-robin fashion, i.e., the i-th interval is assigned to partition $h(i) = (i \mod k)$. This causes the active tuple sets A^R, A^S at each instance of the EBI join to become small, because neighboring intervals are assigned to different partitions. As the cardinality of A^R, A^S impacts the run time of EBI, each join in Line 9 is cheap. On the other hand, the intervals in each partition span the entire domain, meaning that the data in each partition are much sparser compared to the entire dataset. This causes Paradigm 1 to have an increased number of endpoint comparisons compared to a single-threaded algorithm, as k increases. In particular, recall that the basic cost of FS and EBI is the sweeping of the whole space, incurring $|R| + |S|$ and $2 \cdot (|R| + |S|)$ comparisons,
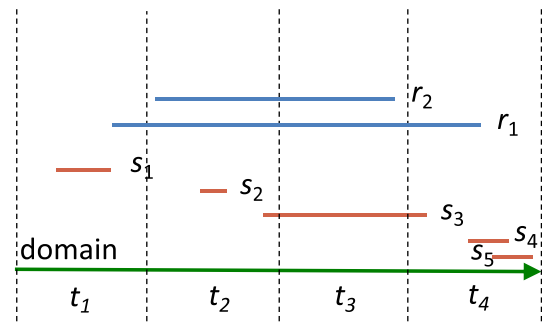
respectively. Under hash-based partitioning, k^2 joins are executed in parallel, and each partition carries $|R|/k + |S|/k$ intervals. Hence, the total basic cost becomes $k \cdot (|R| + |S|)$ and $2 \cdot k \cdot (|R| + |S|)$, respectively (i.e., an increase by a factor of k).

7.3 Domain-based partitioning

Similar to Paradigm 1, our domain-based partitioning paradigm for parallel interval joins (Paradigm 2) involves two phases. The first phase (Lines 1–13) splits the domain uniformly into k non-overlapping stripes; a partition R_j (resp. S_j) is created for each domain stripe t_j. Let t_{start}, t_{end} denote the stripes that cover $r.\text{start}$, $r.\text{end}$ of an interval $r \in R$, respectively. Interval r is first assigned to partition R_{start} created for stripe t_{start}. Then, r is *replicated* across stripes $t_{start+1} \ldots t_{end}$. During the second phase (Lines 15–16), the domain-based paradigm computes $R_j \bowtie S_j$ for every domain stripe t_j, independently. To avoid producing duplicate results, a join result (r, s) is reported if *at least one* of the involved intervals is not a replica. We can easily prove that if for both r and s the start endpoint is not in t_j, then r and s should also intersect in the previous stripe t_{j-1}, therefore (r, s) will be reported by another partition-join.

We show the difference between the two paradigms using Fig. 2; without loss of generality, assume that we are allocating 4 CPU threads for computing $R \bowtie S$. To fully take advantage of parallelism, we assign each partition-join to a separate thread. Hence, the hash-based paradigm will first create $\sqrt{4} = 2$ partitions for each input, i.e., $R_1 = \{r_1\}$, $R_2 = \{r_2\}$ for collection R and $S_1 = \{s_1, s_3, s_5\}$, $S_2 = \{s_2, s_4\}$ for S, and then evaluate pairwise joins $R_1 \bowtie S_1$, $R_1 \bowtie S_2$, $R_2 \bowtie S_1$ and $R_2 \bowtie S_2$. In contrast, the domain-based paradigm will first split the domain into the 4 disjoint stripes pictured in Fig. 9, and then assign and replicate (if needed) the intervals into 4 partitions for each collection; $R_1 = \{r_1\}$, $R_2 = \{\hat{r}_1, r_2\}$, $R_3 = \{\hat{r}_1, \hat{r}_2\}$, $R_4 = \{\hat{r}_1\}$ for R and $S_1 = \{s_1\}$, $S_2 = \{s_2, s_3\}$, $S_3 = \{\hat{s}_3\}$, $S_4 = \{\hat{s}_3, s_4, s_5\}$ for S, where $\hat{r}_j$ (resp. $\hat{s}_j$) denotes the replica of an interval

PARADIGM 2: Domain-based Partitioning

Input	: collections of intervals R and S, number of partitions k
Output	: all intersecting interval pairs $(r, s) \in R \times S$

```
 1 split domain into k stripes;
 2 foreach interval r ∈ R do                        ▷ partition R
 3   │  t_start ← domain stripe covering r.start;
 4   │  t_end ← domain stripe covering r.end;
 5   │  add r to partition R_start;
 6   │  foreach stripe t_j inside (t_start, t_end] do
 7   │  │  └  replicate r to partition R_j;

 8 foreach interval s ∈ S do                        ▷ partition S
 9   │  t_start ← domain stripe covering s.start;
10   │  t_end ← domain stripe covering s.end;
11   │  add s to partition S_start;
12   │  foreach stripe t_j inside (t_start, t_end] do
13   │  │  └  replicate s to partition S_j;

14 foreach domain stripe t_j do
15   │  compute R_j ⋈ S_j;                          ▷ LEBI, FS and variants
```

$r_i \in R$ (resp. $s_i \in S$) inside stripe t_j. Last, the paradigm will compute partition-joins $R_1 \bowtie S_1$, $R_2 \bowtie S_2$, $R_3 \bowtie S_3$ and $R_4 \bowtie S_4$. Note that $R_3 \bowtie S_3$ will produce no results because all contents of R_3 and S_3 are replicas, while $R_4 \bowtie S_4$ will only produce (r_1, s_4) but not (r_1, s_3) which will be found in $R_2 \bowtie S_2$.

Our domain-based partitioning paradigm achieves a higher degree of parallelism compared to Paradigm 1, because for the same number of partitions it requires quadratically fewer joins. Also, as opposed to previous work that also applies domain-based partitioning (e.g., [9,24]), we avoid the production and elimination of duplicate join results. On the other hand, *long lived* intervals that span a large number of stripes and skewed distributions of start endpoints create joins of imbalanced costs. In what follows, we propose two orthogonal techniques that deal with load balancing.

7.3.1 Mini-joins and Greedy scheduling

Our first optimization of Paradigm 2 is based on decomposing the partition-join $R_j \bowtie S_j$ for a domain stripe t_j into a number of *mini-joins*. The mini-joins can be executed independently (i.e., by a different thread) and bear different costs. Hence, they form tasks that can be greedily scheduled based on their cost estimates, in order to achieve load balancing.

Specifically, consider stripe t_j and let t_j.start and t_j.end be its endpoints. We distinguish between the following cases for an interval $r \in R$ (resp. $s \in S$) which is in partition R_j (resp. S_j):

(A) r starts inside t_j, i.e., t_j.start $\leq r$.start $< t_j$.end,

(B) r starts inside a previous stripe but ends inside t_j, i.e., r.start $< t_j$.start and r.end $< t_j$.end, or

(C) r starts inside a previous stripe and ends after t_j, i.e., r.start $< t_j$.start and r.end $\geq t_j$.end.

Note that in cases (B) and (C), r is assigned to partition R_j by replication (Lines 7–8 and 13–14 of Paradigm 2). We use R_j^A, R_j^B, and R_j^C (resp. S_j^A, S_j^B, and S_j^C) to denote the *mini-partitions* of R_j (resp. S_j) that correspond to the 3 cases above.

Under this, we can break partition-join $R_j \bowtie S_j$ down to 9 distinct *mini-joins*; only 5 of these 9 need to be evaluated while the evaluation for 4 out of these 5 mini-joins is simplified. Specifically:

– $R_j^A \bowtie S_j^A$ is evaluated as normal; i.e, as discussed in Sections 3 and 4.

– For $R_j^A \bowtie S_j^B$ and $R_j^B \bowtie S_j^A$, join algorithms only visit end endpoints in S_j^B and R_j^B, respectively; S_j^B and R_j^B only contain replicated intervals from previous stripes which are properly flagged to precede all intervals starting inside t_j, and so, they form the sole group from S_j^B and R_j^B when the grouping optimization technique is used.

– $R_j^A \bowtie S_j^C$ and $R_j^C \bowtie S_j^A$ reduce to cross-products, because replicas inside mini-partitions S_j^C and R_j^C span the entire stripe t_j; hence, all interval pairs are directly output as results without any endpoint comparisons.

– $R_j^B \bowtie S_j^B$, $R_j^C \bowtie S_j^B$, $R_j^C \bowtie S_j^B$, $R_j^C \bowtie S_j^C$ are not executed at all, as intervals from both inputs start in a previous stripe, and hence the results of these mini-joins would be duplicates.

Given a fixed number n of available CPU threads, i.e., partitioning of the domain into $k = n$ stripes, our goal is to assign *each* of the $1 + 5 \cdot (k - 1)$ in total mini-joins [8] to a thread, in order to evenly distribute the load among all threads, or else to minimize the maximum load per thread. This is a well known NP-hard problem, which we opt to solve using a classic $(4/3 - 1/3n)$-approximate algorithm [17] that has very good performance in practice. The algorithm greedily assigns to the CPU thread with currently the lowest load the next largest job. In details, we first estimate the cost of each mini-join; a straightforward approach for this is to consider the product of the cardinalities of the involved mini-partitions. Next, for each available thread p, we define its *bag* b_p that contains the mini-joins to be executed and its load ℓ_p by adding up the estimated cost of the mini-joins in b_p; initially, b_p is empty and $\ell_p = 0$. We organize the bags in a min-priority queue $\mathcal{Q}$ based on their load. Last, we examine all mini-joins in descending order of their estimated cost. For each mini-join say $R_j^A \bowtie S_j^A$, we *remove* bag b_p at the top of $\mathcal{Q}$ corresponding to thread p with the lowest load, we *append* $R_j^A \bowtie S_j^A$ to b_p and *re-insert* the bag to $\mathcal{Q}$. This

---

[8] The only possible mini-join for the first stripe is $R_j^A \bowtie S_j^A$, as it is not possible for it to contain any replicas.

greedy scheduling algorithm terminates after all mini-joins are appended to a bag.

Discussion and implementation details In practice, the greedy scheduling algorithm replaces an *atomic* assignment approach (Lines 15–16 of Paradigm 2) that would schedule each partition-join as a whole to the same thread. The breakdown of each partition-join task into mini-joins that can be executed at different CPU threads greatly improves load balancing in the case where the original tasks have big cost differences.

7.3.2 Adaptive partitioning

Our second *adaptive partitioning* technique for load balancing re-positions the borders between the $\{t_1, \ldots, t_k\}$ stripes, aiming at making the costs of all partition-joins on Line 16 in Paradigm 2 similar. Assuming a 1-1 assignment of partition-joins to CPU threads, load balancing can be achieved by finding the optimal k partitions that minimize the maximum partition-join cost. This can be modeled as the problem of defining a k-bins histogram with the minimum maximum error at each bin. [9] This problem can be solved exactly in PTIME with respect to the domain size, with the help of dynamic programming [20]; however, in our case the domain of the intervals is huge, so we resort to a heuristic that gives a good solution very fast. The time taken for partitioning should not dominate the cost of the join (otherwise, the purpose of a good partitioning is defeated). Our heuristic is reminiscent to local search heuristics for creating histograms in large domains that do not have quality guarantees but compute a good solution in practice within short time [30]. Note that, in practice, the overall execution time is dominated by the most expensive partition-join. Hence, given as input an initial set of stripes and partitions (more details in the next paragraph), we perform the following steps. First, the CPU thread or equivalently the stripe t_j that carries the highest load is identified. Then, we reduce t_j's load (denoted as ℓ_j) by moving consecutive intervals from R_j and S_j to the corresponding partitions of its neighbor stripe with the highest load, i.e., either t_{j-1} or t_{j+1}, until $\ell_{j-1} > \ell_j$ or $\ell_{j+1} > \ell_j$ holds, respectively. Intuitively, this procedure corresponds to advancing endpoint t_j.start or retreating t_j.end. Last, we continuously examine the thread with the highest load until no further moving of the load is possible.

The implementation of this heuristic raises two important challenges; (i) how we can quickly estimate the load on each of the $n = k$ available CPU threads and (ii) what is the smallest unit of load (in other words, the smallest number

of intervals) to be moved in between threads/stripes. To deal with both issues we build histogram statistics H^R and H^S for the input collections *online*, without extra scanning costs. In particular, we create a much finer partitioning of the domain by splitting it to a predefined number ξ of *granules* with ξ being a large multiple of k, i.e., $\xi = \alpha \cdot k$, where $\alpha >> 1$. For each granule g, we count the number of intervals $H^R[g]$ and $H^S[g]$ from R and S, respectively, that start inside g. We define every initial stripe t_j as a set of consecutive α granules; in practice, this partitions the input collections into stripes of equal widths as our original framework. Further, we select a granule as the smallest unit (number of intervals) to be moved between stripes. The load on each thread depends on the cost of the corresponding partition-join. This cost is optimized if we break it down into mini-joins, as described in Sect. 7.3.1, because numerous comparisons are saved. Empirically, we observed that the cost of the entire bundle of the 5 mini-joins for a stripe t_j is dominated by the first mini-join, i.e., $R_j^A \bowtie S_j^A$, the cost of which can be estimated by $|R_j^A| \cdot |S_j^A|$. Hence, in order to calculate $|R_j^A|$ (resp. $|S_j^A|$), we can simply accumulate the counts $H^R[g]$ (resp. $H^S[g]$) of all granules $g \in t_j$. As the heuristic changes the boundaries of a stripe t_j by moving granules to/from t_j, cardinalities $|R_j^A|$, $|S_j^A|$ and the join cost estimate for t_j can be incrementally updated very fast.

8 Strategies for parallel partitioning

We next elaborate on how the partitioning process can benefit from modern parallel hardware. We discuss three strategies applicable on both the hash-based and the domain-based partitioning; in the next section, we carefully evaluate these strategies for each partitioning type. As a common feature, all strategies operate in three phases. During the first phase, all available CPU cores or threads are employed to calculate the cardinality of each $|R_j|$ and $|S_j|$ partition. During the second phase, the threads are employed to allocate the space required to store every partition in main memory and then physically partition the input collections. Finally, again all available threads are used to sort and index (if needed) the input partitions, depending on the interval join algorithm to be used. [10] In the following, we detail the first two phases for each partitioning strategy.

One2One The first strategy was used in [29] for hash-based partitioning but can be straightforwardly applied for the domain-based as well. The idea is to exclusively assign every

---

[9] We assume that there is a function to compute/update the cost of each partition-join in constant time; this function should be monotonic with respect to the sub-domain covered by the corresponding stripe, which holds in our case.

[10] Recall that every partition may take part in multiple joining tasks. Hence, we choose to introduce a separate sorting/indexing phase instead of having this step integrated inside the join algorithm.

STRATEGY 1: One2One

Input	: collection of intervals R, number of partitions k, number of threads c
Output	: partitions $\{R_1, \ldots, R_k\}$
Variables	: counters $\{|R_1|, \ldots, |R_k|\}$

1   **create** c parallel threads;
2   **foreach** *thread* j **do**   ▷ executed in parallel
3   **assign** the j-th set of $\frac{k}{c}$ partitions to the thread;
4   **read** intervals from R;
5   **calculate** counters $\{|R_{\left((j-1)\cdot\frac{k}{c}+1\right)}|, \ldots, |R_{\left(j\cdot\frac{k}{c}\right)}|\}$;
6   **allocate** memory space for assigned partitions;
7   **read** intervals from R;
8   **fill** partitions $\{R_{\left((j-1)\cdot\frac{k}{c}+1\right)}, \ldots, R_{\left(j\cdot\frac{k}{c}\right)}\}$;
9   **return** $\{R_1, \ldots, R_k\}$;

R_j (resp. S_j) partition to a single thread.[11] Under this, the thread executes all phases of the partitioning process for R_j. As every partition of the collection is assigned to exactly one thread, the entire partitioning process is essentially divided into smaller independent tasks which run in parallel without the need of synchronization. Strategy 1 illustrates a high-level pseudo-code of One2One. After initiating c parallel threads in Line 1, every thread executes the first and the second phase of the partitioning independently in Lines 3–8. Consider thread j. During the first phase in Lines 3–5, thread j is assigned $\frac{k}{c}$ partitions for the input collection R, where k is the number of requested partitions and c is the number of available threads. Specifically, the thread gets all partitions in the range from $\left((j-1)\cdot\frac{k}{c}+1\right)$ to $\left(j\cdot\frac{k}{c}\right)$ Then, it scans collection R to count how many intervals will be contained inside its assigned partitions. Last, during the second phase in Lines 6–8, every thread allocates the space needed to store their assigned partitions and then, scans for the second time the input collection to fill these partitions.

Despite its simplicity, the One2One strategy has two important drawbacks. First, it requires multiple scans over the input; to be precise, the collection is scanned $2 \cdot c$ times. Second, the strategy cannot cope with skewed data distributions; essentially, the cost of the entire partitioning process is dominated by the cost of processing the largest partition. In what follows, we discuss two partitioning strategies that address these issues.

Temps The key idea for fast partitioning is to assign parts of the input collection to the available threads instead of entire partitions. Under this, every thread reads a chunk from the input containing $\frac{|R|}{c}$ intervals, and builds a temporary local partitioning. The input chunks should be disjoint such that the parallel threads operate completely independently. Every thread performs a first scan of its assigned intervals to count how large its local partitions will be, then allocates the required space in main memory and reads again the intervals

---

STRATEGY 2: Temps

Input	: collection of intervals R, number of partitions k, number of threads c
Output	: partitions $\{R_1, \ldots, R_k\}$
Variables	: global counters $\{|R_1|, \ldots, |R_k|\}$, local partitions $\{R_1^j, \ldots, R_k^j\}$ and local counters $\{|R_1^j|, \ldots, |R_k^j|\}$ for every parallel thread j

1   **create** c parallel threads;
2   **foreach** *thread* j **do**   ▷ executed in parallel
3   **read** the j-th chunk of $\frac{|R|}{c}$ intervals from R;
4   **calculate** local counters $\{|R_1^j|, \ldots, |R_k^j|\}$;
5   **allocate** memory space for $\{R_1^j, \ldots, R_k^j\}$;
6   **read** the j-th chunk of $\frac{|R|}{c}$ intervals from R;
7   **fill** local partitions $\{R_1^j, \ldots, R_k^j\}$;
8   **wait** until all threads finished;   ▷ synchronization
9   **foreach** *partition* R_i **do**   ▷ executed in parallel
10   **calculate** global counter $|R_i| = \sum_{j=1}^{c} |R_i^j|$;
11   **allocate** memory space;
12      $R_i \leftarrow \bigcup_{j=1}^{c} R_i^j$;   ▷ unify local partitions
13   **return** $\{R_1, \ldots, R_k\}$;

---

to fill the partitions. Finally, after all threads have finished, the local partitionings are unified into the final result as the last step.

Strategy 2 illustrates a high-level pseudo-code of Temps. In Lines 2–7, every thread scans (two times) its assigned chunk of the input collection to create a local partitioning. Specifically, thread j gets the j-th chunk of $\frac{|R|}{c}$ input intervals and produces local partitioning $\{R_1^j, \ldots, R_k^j\}$; notice that local partitionings contain the same number of partitions as the final result. To count the cardinality of its local partitions, the thread maintains private local counters $\{|R_1^j|, \ldots, |R_k^j|\}$. After all local partitionings are built (synchronization barrier in Line 8), Temps unifies them by copying local partitions to a contiguous space allocated in main memory for the final partitions, in Lines 9–12. Both the hash-based and the domain-based partitioning assign every interval to exactly one local partition; the same holds for the replicas in case of domain-based. Under this, the cardinality for each final partition R_i is calculated as $|R_i| = \sum_{j=1}^{c} |R_i^j|$ and the partition is defined as $R_i^1 \bigcup \ldots \bigcup R_i^c$, where c is the total number of parallel threads and local partitionings. Last, to accelerate this unification step, the Temps strategy assigns the computation of every partition R_i to the next available thread in a round robin fashion.

Compared to One2One, the Temps strategy scans the entire input collection R only twice as every thread now operates on a different chunk of R. In addition, as R's chunks are equi-sized, i.e., all contain at most $\frac{|R|}{c}$ intervals, the partitioning load is better distributed to the available threads. But, Temps still exhibits important shortcomings. First, for every partition R_i, the strategy allocates twice the required space in main memory, i.e., to store both its corresponding local partitions and R_i itself. Second, the strategy introduces an extra costly step, i.e., the unification of local partitioning.

---

[11] In general, the number of partitions per input may exceed the number of available threads in which case, every thread is responsible for multiple partitions.

STRATEGY 3: Divs

Input	: collection of intervals R, number of partitions k, number of threads c
Output	: partitions $\{R_1, \ldots, R_k\}$
Variables	: global counters $\{|R_1|, \ldots, |R_k|\}$, and local counters $\{|R_1^j|, \ldots, |R_k^j|\}$ for every parallel thread j

1   **create** c parallel threads;
2   **foreach** *thread* j **do**   ▷ executed in parallel
3   **read** the j-th chunk of $\frac{|R|}{c}$ intervals from R;
4   **calculate** local counters $\{|R_1^j|, \ldots, |R_k^j|\}$;
5   **wait** until all threads finished;   ▷ synchronization
6   **foreach** *partition* R_i **do**   ▷ executed in parallel
7   **calculate** global counter $|R_i| = \sum_{j=1}^{c} |R_i^j|$;
8   **allocate** memory space;
9   **divide** partition into c logical parts;
10 **wait** until all threads finished;   ▷ synchronization
11 **foreach** *thread* j **do**   ▷ executed in parallel
12   **read** the j-th chunk of $\frac{|R|}{c}$ intervals from R;
13   **fill** j-th part of each partition in $\{R_1, \ldots, R_k\}$;
14 **return** $\{R_1, \ldots, R_k\}$;

Also, the cost of this last step is dominated by the largest partition which is again computed by a single thread.

Divs To address these shortcomings, we next discuss our last strategy. Strategy Divs shares the same key idea to Temps, i.e., every thread j processes independently the j-th chunk of $\frac{|R|}{c}$ input intervals. But, instead of building a temporary local partitioning, the thread directly updates the final partitions. For this purpose, the strategy logically divides every final partition R_i into c parts, i.e., one for each available thread. The extent of each R_i^j part is determined by local counters $|R_i^j|$, which are computed similar to strategy Temps. With this division, each thread independently fills a dedicated part of R_i's data structure in memory without the need of locking or any type of synchronization.

Strategy 3 illustrates a high-level pseudo-code of Divs. Lines 2 and 3 are identical to Strategy 2, i.e., a first scan of the input collection determines local counters $\{|R_1^j|, \ldots, |R_k^j|\}$ for each thread j. After local counters are computed (synchronization barrier in Line 5), Divs allocates the necessary space in main memory to build every R_i partition (Lines 7–8) and also, logically divides R_i into c parts using its local counters (Line 9). Finally after this preparation step is finished for all partitions (synchronization barrier in Line 10), every thread scans for the second time its assigned input intervals and fills its dedicated part inside the data structure of every partition, in Lines 10–13.

Compared to Temps, the Divs strategy does not allocate extra space for every partition; at the same time, the costly unification step of Temps is entirely avoided. In addition, the largest partition which could become the bottleneck for both strategies One2One and Temps is now filled by multiple threads in parallel achieving a better load balancing.

9 Experiments on parallel processing

Last, we present the second part of our experimental evaluation, which focuses on the parallel computation of interval joins. In view of the results for single-threaded processing in Sect. 6, we next focus on optFS.

9.1 Setup

The experiments were conducted on the same machine used for the single-threaded tests in Sect. 6 with an identical setup, i.e., XOR workload, all data stored in main memory. Further, we chose to activate hyper-threading which allowed us to run up to 40 threads and used OpenMP for multi-threaded processing. Besides varying the $|R|/|S|$ ratio inside $\{0.25, 0.5, 0.75, 1\}$, we also increase the number of available parallel threads inside $\{5, 10, 15, 20, 25, 30, 35, 40\}$. We indicate the activation of hyper-threading by an h subscript, e.g., 25_h. Last, for the adaptive partitioning, we conducted a series of tests to determine the multiplicative factor α which controls the number of granules in the fine partitioning of the domain (see Sect. 7.3.2). To avoid significantly increasing the partitioning cost, we ended up setting $\alpha = 1000$ when the number of threads is less than 10, and $\alpha = 100$ otherwise.

9.2 Tuning hash-based partitioning

We first tune the hash-based paradigm. [29] sorts every collection prior to partitioning. We experimented with a variant of the paradigm which does not include such a pre-sort step and proved always faster. Hence, in the following we run our variant of the hash-based paradigm. Our analysis investigates which is the best strategy for the parallel partitioning of the inputs and how to select the number of partitions to be created.

9.2.1 Partitioning strategies

Fig. 10 reports the partitioning time of the One2One, Temps and Divs strategies while varying the number of partitions on our six datasets. For all tests, we set $|R| = |S|$ and used up to 20 parallel threads to partition the input collections. The results clearly show that Divs is both the most efficient and the most robust partitioning strategy, i.e., its time is little affected by the increase in the number of partitions. One2One is competitive to Divs only if each collection is split into 20 or more partitions. Recall that One2One assigns each partition to exactly one thread, so with less than 20 partitions, some of the 20 available threads are never used. A key factor for understanding the differences in the performance of the strategies is the size of the inputs (see Table 1). GREEND and TAXIS contain more than 100m intervals; for these datasets,
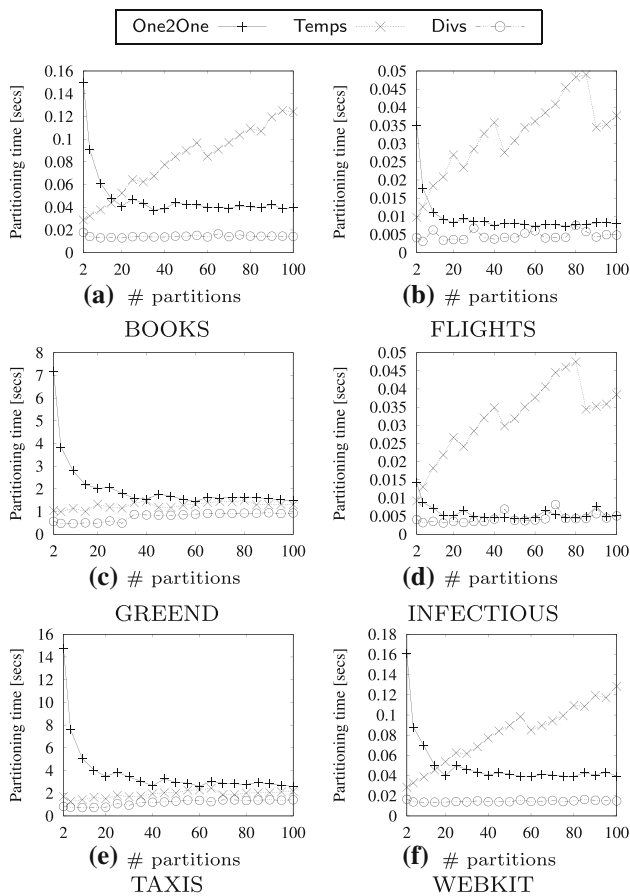
Fig. 10 Tuning hash-based partitioning: strategies, $|R| = |S|$ and 20 threads

Fig. 11 Tuning hash-based partitioning: # partitions, $|R| = |S|$ and 20 threads

One2One is always slower than both Temps and Divs due to scanning these big inputs multiple times while Temps is always slower than Divs due to creating and unifying local partitions. The rest of the datasets contain 2m or less intervals. Provided that at least 20 partitions are created, One2One is always faster than Temps because these partitions contain very few intervals and the overhead from local partitioning in Temps becomes increasingly higher by the number of partitions.

9.2.2 Number of partitions

Piatov et al. [29] suggested that the hash-based paradigm performs at its best when each input is split into $\sqrt{n}$ partitions, where n is the number of available threads. Under this, every available thread is assigned exactly one of the n in total partition-joins. Although we used this heuristic in our preliminary work [5], we investigate here in detail the impact of the number of partitions.

Figure 11 reports the breakdown of optFS execution time while varying the number of partitions in each collection from 1 to 1, 000; note that the number of available parallel threads
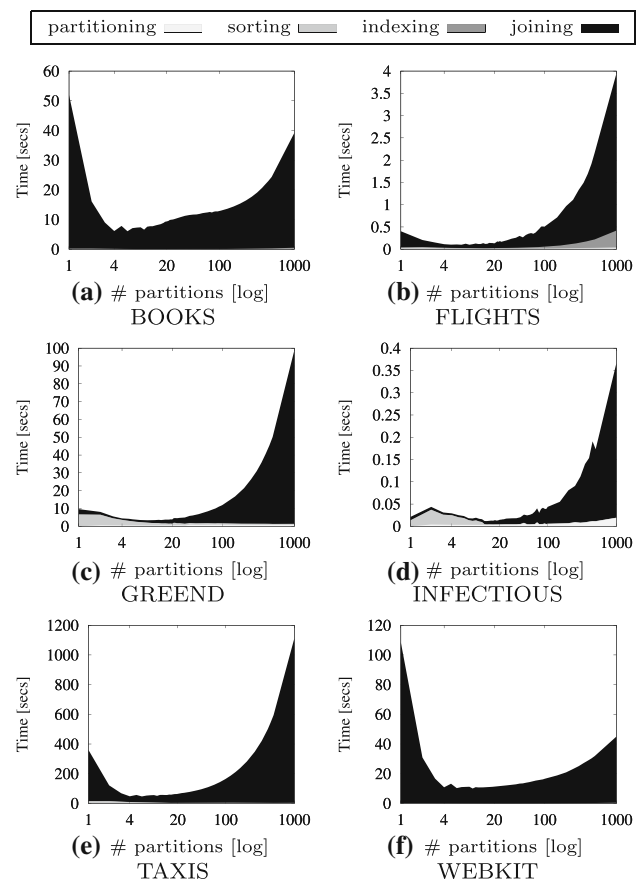
is fixed to 20. As expected, there is a tradeoff between the number of partitions and the total execution time. Initially, optFS benefits from splitting each input into more partitions but the algorithm slows down when the number of partitions exceeds a particular value. However, our tests also unveil a correlation between the number of partitions and the selectivity of the join. For the highly selective queries in GREEND and INFECTIOUS, the execution time of optFS is minimized when the number of partitions equals almost the number of available threads. On the other hand, for queries of low or medium selectivity, the heuristic from [29] is effective, i.e., the number of partitions should be set to $\lfloor\sqrt{20}\rfloor = 4$. To understand this behavior, observe the time breakdown in Figures 11(c) and (d) when the number of partitions is set below 20, especially equal to 4. Different from all other cases, the total execution time is dominated by the sorting cost; the actual joining phase is very cheap due to the low number of results. Essentially, we can enhance sorting by splitting the inputs into more partitions which creates smaller sorting tasks to run in parallel.
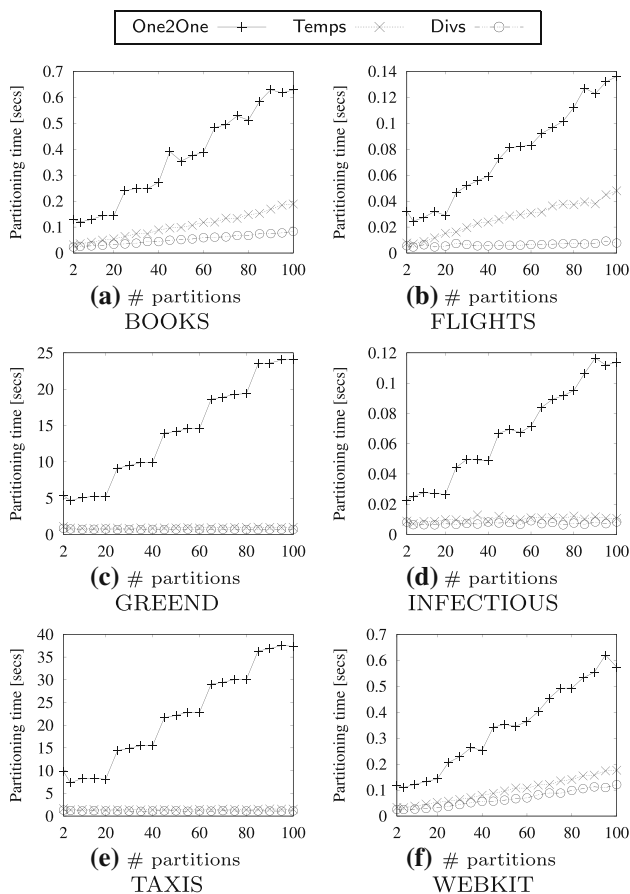
Fig. 12 Tuning domain-based partitioning: strategies, $|R| = |S|$ and 20 threads

9.3 Tuning domain-based partitioning

We next tune our domain-based paradigm. Besides determining the best strategy for parallel partitioning and the number of partitions, we also study the impact of our load balancing techniques from Sect. 7.3.

9.3.1 Partitioning strategies

Fig. 12 reports the domain-based partitioning time for strategies One2One, Temps and Divs while varying the number of partitions; for the tests, we set again $|R| = |S|$ and used up to 20 parallel threads. Also, adaptive partitioning from Sect. 7.3.2 was deactivated. Similar to Sect. 9.2.1, we observe that Divs is the most efficient and most robust strategy for parallel partitioning; on the largest datasets GREEND and TAXIS, Temps is competitive to Divs but still slower. However, different to our hash-based analysis, One2One is clearly the slowest strategy in all cases; its time is severely affected by the increase in the number of partitions exhibiting also a "staircase" pattern (more obvious in Figures 12(c) and (e)). The difference in One2One's behavior is due to the higher

processing cost per interval incurred by the domain-based partitioning compared to hash-based. This cost is amplified by the increase in the number of partitions. Recall that for hash-based partitioning, we only need to hash the start endpoint of every interval. In contrast, for domain-based partitioning we also need to replicate an interval to all overlapping stripes; the replication cost naturally increases with the number of partitions. Regarding the "staircase" pattern, notice that One2One's time essentially goes up every 20 partitions. Consider for example the increase from 20 to 40 partitions. At first, every thread builds exactly one partition. When we increase the number of partitions to 21, this extra partition will be assigned as a second task to one of the available threads. The total time of this thread will increase and dominate the overall partitioning time Adding more partitions will not change this overall time because there still threads assigned one partition unless the total number of partitions grows higher than 40.

9.3.2 Number of partitions

In [5], we always set the number of partitions equal to the number of threads such that each thread is assigned exactly one partition-join. To confirm the effectiveness of this heuristic, we measure the runtime of optFS under the domain-based paradigm while varying the number of partitions from 1 to 1, 000. Similar to Sect. 9.2.2, the number of available threads is set to 20.

Figure 13 reports the results of our tests. The expected tradeoff between the execution time and the number of partitions from each collection is again observed. But, different from the hash-based paradigm, optFS under the domain-based performs at its best when the number of partitions equals the number of available threads. An exception arises for the very selective joins; in INFECTIOUS, the lowest execution time is observed for around 100 partitions per input while in GREEND for over 100. Nevertheless, we can safely use the same heuristic even in these cases because (i) the average execution time for INFECTIOUS joins is extremely low (below 20 msec) even for 20 partitions while (ii) for GREEND, the time does not significantly drop when the number of partitions exceeds 20.

9.3.3 Load balancing

We now evaluate the load balancing achieved by the optimizations of domain-based partitioning of Section 7.3. To save space, we only show the results on WEBKIT; similar conclusions can be drawn for join queries on the other datasets. Apart from the overall execution time of each join, we also measured the load balancing among the participating CPU threads. Let set $L = \{\ell_1 \ldots \ell_n\}$ be the measured time spent by each of the available n threads; we define the
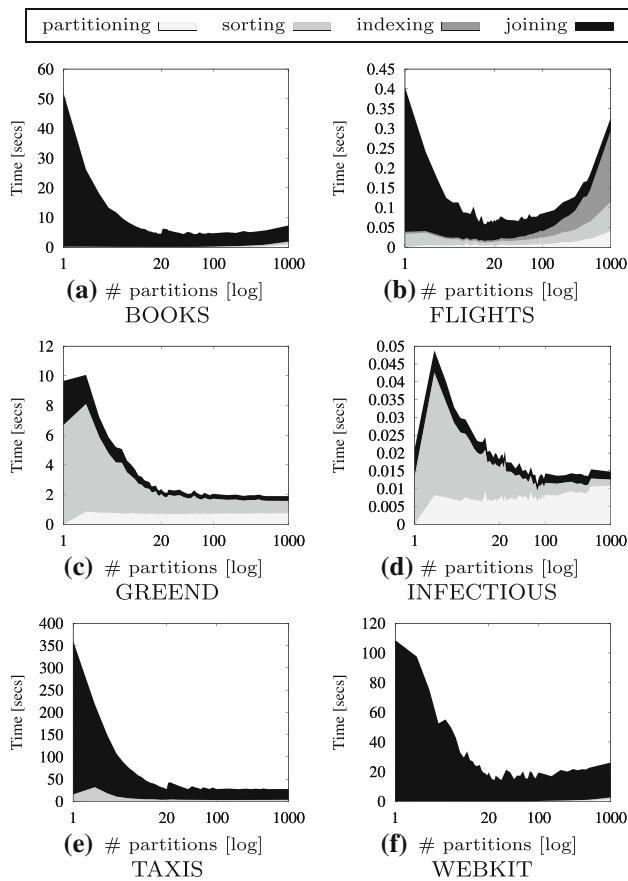
Fig. 13 Tuning domain-based partitioning: # partitions, $|R| = |S|$ and 20 threads

Fig. 14 Tuning domain-based partitioning: load balancing, optFS on WEBKIT

average idle time as:

$$\frac{1}{n} \sum_{j=1}^{n} \{max(L) - \ell_j\}$$

A high average idle time means that the threads are under-utilized in general, whereas a low average idle time indicates that the load is balanced.

We experimented by activating or deactivating the mini-joins breakdown denoted by *mj* (Section 7.3.1), greedy scheduling denoted by *greedy* (Sect. 7.3.1), and adaptive partitioning denoted by *adaptive* (Section 7.3.2). We use the term *atomic* to denote the assignment of each partition-join or the bundle of its corresponding 5 mini-joins to the same thread, and *uniform* to denote the (non-adaptive) uniform initial partitioning of the domain. We tested the following setups: [12]

(1) *uniform/atomic* is the baseline domain-based para-digm of Sect. 7.3 with all load balancing optimization techniques deactivated;

(2) *atomic/adaptive* is an extension to the baseline that employs only the adaptive partitioning;

(3) *uniform/mj+atomic* splits each partition-join of the baseline into 5 mini-joins which are all executed by the same CPU thread;

(4) *adaptive/mj+atomic* first applies the adaptive partitioning technique and then splits each partition-join into 5 mini-joins to be all executed by the same thread;

(5) *uniform/mj+greedy* splits each partition-join of the baseline into 5 mini-joins which are greedily distributed to the available threads;

(6) *adaptive/mj+greedy* employs all optimizations.

Figures 14(a), (c) report the total execution time for each setup (1)–(6), while Figures 14(b), (c) report the ratio of the average idle time over the execution time.

We observe the following. First, setups (2)–(6) all manage to enhance the parallel computation of the join. Their execu-

---

[12] Based on our analysis in Sect. 9.3.2, *greedy/uniform* or *greedy/adaptive* setups are meaningless since the number of partitions equals the number of available CPU threads.

Table 7 Setups for partitioning-based computation

		Hash-based	Domain-based
Partitioning	# partitions	uFS: # threads	# threads
		bgudFS: $\lfloor \sqrt{\# \, threads} \rfloor$	
	Strategy	Divs	Divs
	Adaptive	–	Yes
Joining	Mini-joins Breakdown	–	Yes
	Greedy Scheduling	–	Yes

tion time is lower than the time of the *uniform/atomic* baseline. The most efficient setups always include the *mj+greedy* combination regardless of activating adaptive partitioning or not. In practice, splitting every partition-join into 5 mini-joins creates mini-jobs of varying costs (recall that 2 of them are cross-products and other 2 are also quite cheap), which facilitates the even partitioning of the total join cost to processors. For example, if one partition is heavier overall compared to the others, one thread would be dedicated to its most expensive mini-join and the other mini-joins would be handled by less loaded CPU threads. Also, notice that the *mj* optimization is beneficial even when the 5 defined mini-joins are all executed by the same CPU thread (i.e., *uniform/mj+atomic*), although the benefit is small compared to the other setups. This is because breaking down a partition-join into 5 mini-joins greatly reduces the overall cost of the partition-join (again, recall that 4 of the mini-joins are cheap).

Adaptive partitioning appears to have a smaller impact compared to the other two optimizations. Among the setups that do not employ the *greedy* scheduling, *adaptive/atomic* ranks first (both in terms of the execution time the average idle time ratio) but when activated on top of the *uniform/mj+greedy* setup, adaptive partitioning enhances the join computation when the number of threads is low, below 20; notice how faster is the *adaptive/mj+greedy* setup compared to *uniform/mj+greedy* in case of 5 available CPU threads.

Overall, we observe that (i) the *mj* optimization greatly reduces the cost of a partition-join and adds flexibility in load balancing, (ii) the *uniform/mj+greedy* and *adaptive/mj+greedy* setups perform very well in terms of load balancing, by reducing the average idle time of any thread to below 20% of the total execution time in almost all cases ($|R|/|S| = 0.25$ and when less than 15 threads are available for *uniform/mj+greedy* are the only exceptions).

9.4 Comparisons

Table 7 summarizes the best setup for optFS under the hash-based and the domain-based paradigms. Both paradigms use
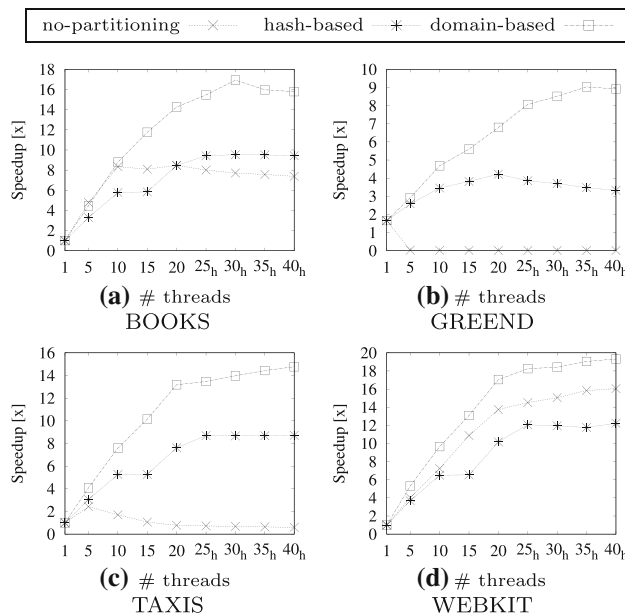

Fig. 15 Comparing parallel processing solutions: optFS speedup for $|R| = |S|$

Divs to efficiently partition the inputs. For hash-based, we set the number of partitions on the selectivity of the join, i.e., depending on whether optFS acts as uFS or bgudFS; for domain-based, we always set the number of partitions equal to the number of available CPU threads. Also, to take full advantage of all proposed load balancing optimizations, we setup the domain-based paradigm as *adaptive/mj+greedy*.

We next compare all three approaches for the parallel computation of interval joins. [13] We first report in Fig. 15 the speedup over the single-threaded optFS (Sect. 6), while varying the number of available CPU threads; to save space, we omit the results on FLIGHTS and INFECTIOUS since the findings are identical to TAXIS and GREEND, respectively. Overall, we observe that the domain-based paradigm is clearly the most efficient approach, being able to achieve

---

[13] We also tested a hybrid that applies domain-based partitioning and uses no-partitioning for every partition-join, but, this approach was always slower than original no-partitioning.

the highest speedup in all cases. In fact, the performance advantage of the domain-based paradigm grows by the number of available threads. This is because the queries benefit increasingly more from domain-based's ability to significantly reduce the number of endpoint comparisons conducted. In contrast, the number of comparisons under the hash-based paradigm increases, compared even to single-threaded optFS, as the number of available threads goes up.[14] Our tests also reveal the role of join selectivity. For the highly selective queries in GREEND and INFECTIOUS, the hash-based paradigm always outperforms no-partitioning, but for the low selectivity joins in BOOKS and WEBKIT, no-partitioning is competitive; in fact, for WEBKIT, it achieves always the second highest speedup. For queries of medium selectivity, i.e., in FLIGHTS and TAXIS, no-partitioning is able to incur a speedup only when up to 5 parallel threads are employed. To understand the behavior of no-partitioning optFS, we need to discuss two important shortcomings stemming from its master-slaves approach. The first problem is thread starvation; essentially, the master thread cannot create forward scan tasks fast enough for the slaves to run. This is the case with highly selective queries, where the forward scans are too short and hence cheap, as Fig. 6 shows. The second problem is the high number of cache misses incurred by all threads scanning the same data structures in main memory. This problem is amplified when increasing the number of CPU threads used as slaves.

Finally, we report in Fig. 16 the total execution time for each approach while varying the $|R|/|S|$ ratio of the input collections; for these tests, we used up to 20 threads. As expected all approaches are affected by increasing the input size; their execution time rises. Nevertheless, the domain-based paradigm outperforms both the hash-based and no-partitioning in every test.

10 Conclusions and future work

In this paper, we targeted the efficient in-memory computation of interval overlap joins. Under single-threaded evaluation, we studied FS, a simple and efficient algorithm based on plane sweep that does not rely on any special data structures. We proposed four novel optimizations for FS that greatly accelerate the algorithm in practice. Our experimental analysis showed that a self-tuning version of FS which automatically selects and applies the most appropriate optimizations is competitive or even faster than the state-of-the-art. For parallel join evaluation, we proposed (i) a master-slaves approach that does not physically partition the inputs and (ii) a domain-based partitioning computation

---

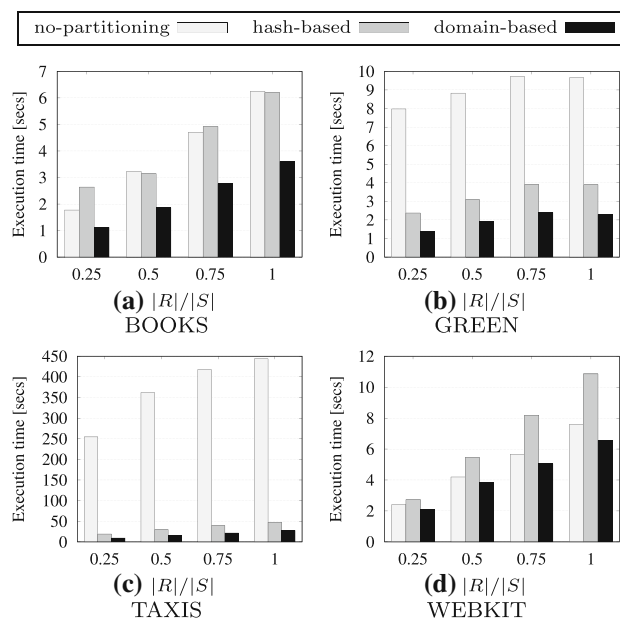[14] Results on endpoint comparisons can be found in our preliminary analysis [5].

Fig. 16 Comparing parallel processing solutions: optFS running time for 20 threads

framework. Under the latter, each partition-join is broken down to five independent mini-joins which can be greedily assigned to the available CPU threads achieving a high degree of load balancing. Our experiments showed that our domain-based partitioning framework for parallel joins significantly outperforms both our no-partitioning approach and the hash-based framework of [29] while also scaling well with the number of available threads. In the future, we plan to study interval joins in stream processing. Also, we intend to investigate novel indexing structures for interval queries and joins.

References

1. Aho, A.V., Ullman, J.D.: Principles of Compiler Design. Addison-Wesley Longman, Boston (1977)
2. Arge, L., Procopiuc, O., Ramaswamy, S., Suel, T., Vitter, J.S.: Scalable sweeping-based spatial join. In: VLDB (1998)
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. Blanas, S., Li, Y., Patel, J.M.: Design and evaluation of main memory hash join algorithms for multi-core cpus. In: SIGMOD (2011)
5. Bouros, P., Mamoulis, N.: A forward scan based plane sweep algorithm for parallel interval joins. PVLDB **10**(11), 1346–1357 (2017)
6. Bouros, P., Mamoulis, N.: Interval count semi-joins. In: EDBT (2018)
7. Brinkhoff, T., Kriegel, H., Seeger, B.: Efficient processing of spatial joins using r-trees. In: SIGMOD (1993)
8. Cafagna, F., Böhlen, M.H.: Disjoint interval partitioning. VLDB J. **26**(3), 447–466 (2017)
9. Chawda, B., Gupta, H., Negi, S., Faruquie, T.A., Subramaniam, L.V., Mohania, M.K.: Processing interval joins on map-reduce. In: EDBT (2014)
10. Chekol, M.W., Pirrò, G., Stuckenschmidt, H.: Fast interval joins for temporal SPARQL queries. In: WWW (2019)
11. Cheng, R., Singh, S., Prabhakar, S., Shah, R., Vitter, J.S., Xia, Y.: Efficient join processing over uncertain data. In: CIKM (2006)
12. Copeland, G.P., Khoshafian, S.: A decomposition storage model. In: SIGMOD (1985)
13. Dignös, A., Böhlen, M.H., Gamper, J.: Overlap interval partition join. In: SIGMOD (2014)
14. Dittrich, J., Seeger, B.: Data redundancy and duplicate detection in spatial join processing. In: ICDE, pp. 535–546 (2000)
15. Enderle, J., Hampel, M., Seidl, T.: Joining interval data in relational databases. In: SIGMOD (2004)
16. Gao, D., Jensen, C.S., Snodgrass, R.T., Soo, M.D.: Join operations in temporal databases. VLDB J. **14**(1), 2–29 (2005)
17. Graham, R.L.: Bounds on multiprocessing timing anomalies. SIAM J. Appl. Math. **17**(2), 416–429 (1969)
18. Gunadhi, H., Segev, A.: Query processing algorithms for temporal intersection joins. In: ICDE (1991)
19. Isella, L., Stehlé, J., Barrat, A., Cattuto, C., Pinton, J.F., den Broeck, W.V.: What's in a crowd? Analysis of face-to-face behavioral networks. J. Theor. Biol. **271**(1), 166–180 (2011)
20. Jagadish, H.V., Koudas, N., Muthukrishnan, S., Poosala, V., Sevcik, K.C., Suel, T.: Optimal histograms with quality guarantees. In: VLDB (1998)
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: SIGMOD (2013)
22. Kriegel, H., Kunath, P., Pfeifle, M., Renz, M.: Distributed intersection join of complex interval sequences. In: DASFAA (2005)
23. Kriegel, H., Pötke, M., Seidl, T.: Managing intervals efficiently in object-relational databases. In: VLDB (2000)
24. Leung, T.Y.C., Muntz, R.R.: Temporal query processing and optimization in multiprocessor database machines. In: VLDB (1992)
25. 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 (2014)
26. Moon, B., López, I.F.V., Immanuel, V.: Efficient algorithms for large-scale temporal aggregation. TKDE **15**(3), 744–759 (2003)
27. Nicolau, A.: Loop quantization: Unwinding for fine-grain parallelism exploitation. Tech. Rep. TR85-709, Dept. of Computer Science, Cornell University (1985)
28. Petersen, W.P., Arbenz, P.: Introduction to Parallel Computing. Oxford Press, Oxford (2004)
29. Piatov, D., Helmer, S., Dignös, A.: An interval join optimized for modern hardware. In: ICDE (2016)
30. Poosala, V., Ioannidis, Y.E., Haas, P.J., Shekita, E.J.: Improved histograms for selectivity estimation of range predicates. In: SIGMOD (1996)
31. Preparata, F.P., Shamos, M.I.: Computational Geometry—An Introduction. Texts and Monographs in Computer Science. Springer, Berlin (1985)
32. Segev, A., Gunadhi, H.: Event-join optimization in temporal relational databases. In: VLDB (1989)
33. Sitzmann, I., Stuckey, P.J.: Improving temporal joins using histograms. In: DEXA (2000)
34. Soo, M.D., Snodgrass, R.T., Jensen, C.S.: Efficient evaluation of the valid-time natural join. In: ICDE (1994)
35. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E.J., O'Neil, P.E., Rasin, A., Tran, N., Zdonik, S.B.: C-store: a column-oriented DBMS. In: VLDB (2005)
36. Tsitsigkos, D., Bouros, P., Mamoulis, N., Terrovitis, M.: Parallel in-memory evaluation of spatial joins. In: SIGSPATIAL (2019)
37. Zhang, D., Tsotras, V.J., Seeger, B.: Efficient temporal join processing using indices. In: ICDE (2002)