A Forward Scan based Plane Sweep Algorithm for Parallel Interval Joins

Panagiotis Bouros*
Department of Computer Science
Aarhus University, Denmark
pbour@cs.au.dk

Nikos Mamoulis†
Dept. of Computer Science & Engineering
University of Ioannina, Greece
nikos@cs.uoi.gr

ABSTRACT

The interval join is a basic operation that finds application in temporal, spatial, and uncertain databases. Although a number of centralized and distributed algorithms have been proposed for the efficient evaluation of interval joins, classic *plane sweep* approaches have not been considered at their full potential. A recent piece of related work proposes an optimized approach based on plane sweep (PS) for modern hardware, showing that it greatly outperforms previous work. However, this approach depends on the development of a complex data structure and its parallelization has not been adequately studied. In this paper, we explore the applicability of a largely ignored *forward scan* (FS) based plane sweep algorithm, which is extremely simple to implement. We propose two optimizations of FS that greatly reduce its cost, making it competitive to the state-of-the-art single-threaded PS algorithm while achieving a lower memory footprint. In addition, we show the drawbacks of a previously proposed hash-based partitioning approach for parallel join processing and suggest a domain-based partitioning approach that does not produce duplicate results. Within our approach we propose a novel breakdown of the partition join jobs into a small number of independent mini-join jobs with varying cost and manage to avoid redundant comparisons. Finally, we show how these mini-joins can be scheduled in multiple CPU cores and propose an adaptive domain partitioning, aiming at load balancing. We include an experimental study that demonstrates the efficiency of our optimized FS and the scalability of our parallelization framework.

1. INTRODUCTION

Given a 1D discrete or continuous space, an interval is defined by a start and an end point in this space. For example, given the space of all non-negative integers $\mathbb{N}$, and two integers $\mathsf{start}, \mathsf{end} \in \mathbb{N}$, with $\mathsf{start} \leq \mathsf{end}$, we define an interval $i = [\mathsf{start}, \mathsf{end}]$ as the subset of $\mathbb{N}$, which includes all integers x with $\mathsf{start} \leq x \leq \mathsf{end}$.[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., $r.\mathsf{start} \leq s.\mathsf{start} \leq r.\mathsf{end}$ or $s.\mathsf{start} \leq r.\mathsf{start} \leq s.\mathsf{end}$.

The interval join is one of the most widely used operators in temporal databases [9]. 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 instance, assume that the employees of a company may be employed at different departments during different time periods. Given the employees who have worked in departments A and B, the interval join would identify pairs of employees, whose periods of work in A and B, respectively, interest.

Interval joins apply in other domains as well. In multidimensional spaces, an object can be represented as a set of intervals in a space-filling curve. The intervals correspond to the subsequences of points on the curve that are included in the object. Spatial joins can then be reduced to interval joins in the space-filling curve representation [14]. 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 [1, 3]. 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 [6].

Due to its wide applicability, there has been quite a number of studies on the efficient evaluation of the interval join. Surprisingly, the use of the classic *plane sweep* (PS) algorithms [20] has not been considered as a competitive approach in most of the previous work.[2] A recent paper [18] implemented and optimized a version of PS (taken from [1]), 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, the algorithm maintains the *active sets* of intervals from R and S which intersect with the current stop point of the line. When the line is at a start point (e.g., from R) the current interval is added to the corresponding active set (e.g., A^R) and the active set of the other relation (e.g., A^S of S) is scanned to form join pairs with the current interval. When the line is at an end point (e.g.,

*Funded by Innovation Fund Denmark as part of the Future Cropping project (J. nr. 5107-00002B).
†Funded from the European Union's Horizon 2020 research and innovation programme under grant agreement No 657347.

Proceedings of the VLDB Endowment, Vol. 10, No. 11

[1] Note that the intervals in this paper are *closed*. Yet, our techniques and discussions are applicable with minor changes for generic intervals where the begin and end sides are either open or closed.
[2] We believe the main reason is that previous work mostly focused on centralized evaluation on hard-disk, which becomes less relevant in today's in-memory data management and the wide availability of parallel and distributed platforms and models.

from R), the corresponding interval is removed from the respective active set (e.g., A^R).

The work of [18] focuses on minimizing the random memory accesses due to the updates and scans of the active sets. However, random accesses can be overall avoided by another implementation of PS, presented in [3] in the context of 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 start point before the end point of r form join results with r. It can be easily shown that the cost of FS (excluding sorting) is $O(|R| + |S| + K)$, where K is the number of join results.

The contribution of this paper is twofold. In Section 4, we present two novel optimizations for FS, which greatly reduce the number of comparisons during the join computation. In particular, optimized FS produces multiple join tuples in batch at the cost of a single comparison. Hence, we achieve (i) competitive performance to the state-of-the-art PS algorithm (EBI [18]), without using any special hardware optimizations and (ii) a much lower memory footprint.

Our second contribution (Section 5) is an optimized framework for processing plane sweep based algorithms in parallel. We first show that the hash-based partitioning framework suggested in [18] does not take full advantage of parallelism. Our framework, applies a domain-based partitioning instead. We first show that although intervals should be replicated in the domain partitions to ensure correctness, duplicate results can be avoided, therefore the partition join jobs can become completely independent. Then, we show how to break down each partition join into five independent *mini-join* jobs which have varying costs. More importantly, 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 a smaller number of CPU cores. In addition, we suggest an adaptive splitting approach for the data domain that results in an improved cost balancing between the partitions and consequently an improved load balancing for the mini-joins. We conduct experiments which show that our domain-based partitioning framework achieves ideal speedup with the number of CPU cores, greatly outperforming the hash-based partitioning framework of [18]. Although our framework is independent of the algorithm used for the mini-joins, we show that our optimized version of FS takes better advantage of it compared to EBI.

The rest of the paper is organized as follows. Section 2 discusses related work while Section 3 reviews in more detail plane sweep methods; EBI [18] and original FS from [3]. In Section 4, we propose two novel optimizations for FS that greatly reduce the computational cost of the algorithm in practice. Section 5 presents our domain-based partitioning framework for parallel interval joins. Section 6 includes our experimental evaluation which demonstrates the effect of our optimizations to FS and the efficiency of our parallel interval join framework. Finally, Section 7 concludes the paper.

2. RELATED WORK

In this section, we review related work on interval joins. We classify the algorithms of previous work based on the data structures they use and based on the underlying architecture.

Nested loops and merge join. Early work on interval joins [11, 21] 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, however, join computation is sub-optimal compared to FS, which guarantees at most $|R|+|S|$ endpoint comparisons that do not produce results.

Index-based algorithms. Enderle et al. [8] propose interval join algorithms, which operate on two RI-trees [15] that index the input collections. Zhang et al. [24] 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 [11, 21]), proposing an extension of the multi-version B-tree [2].

Partitioning-based algorithms. A partitioning-based approach for interval joins was proposed in [23]. 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 [22]. A more sophisticated partitioning approach, called Overlap Interval Partitioning (OIP) Join [7], 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 [8] and sort-merge join. These results are consistent with the comparative study of [9], which shows that partitioning-based methods are superior to nested loops and merge join approaches.

Disjoint Interval Partitioning (DIP) [4] 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 [17]. 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 [17] 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 that are 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 (Section 5), however, the details differ due to the different natures of temporal join and aggregation.

Methods based on plane sweep. The Endpoint-Based Interval (EBI) Join (reviewed in detail in Section 3.1) and its lazy version LEBI were shown to significantly outperform OIP [7] and to also be superior to another plane sweep implementation [1]. An approach similar to EBI is used in SAP HANA [13]. To our knowledge, no previous work was compared to FS [3] (detailed in Section 3.2). In Section 4, we propose two novel optimizations for FS that greatly improve its performance, making it competitive to LEBI.

Parallel algorithms. A domain-based partitioning strategy for interval joins on multi-processor machines was proposed in [16]. Each partition is assigned to a processor and intervals are replicated to the partitions they overlap, in order to ensure that join results can be produced independently at each processor. However, a merge

ALGORITHM 1: Endpoint-Based Interval (EBI) Join

```
   Input      : collections of intervals R and S
   Output     : set J of all intersecting interval pairs (r, s) ∈ R × S
   Variables  : endpoint indices EI^R and EI^S, active interval sets A^R and A^S
1  J ← ∅, 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                 J ← J ⋃ {(r, s)};                  ▷ update results
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                 J ← J ⋃ {(r, s)};                  ▷ update results
22         else
23             remove s from A^S;                     ▷ s no longer open
24         e^S ← next index tuple in EI^S;
25 return J
```

phase with duplicate elimination is required because the same join result can be produced by different processors. Our parallel join processing approach (Section 5) also applies a domain-based partitioning but does not produce duplicates. In addition, we propose a breakdown of each partition join to a set of mini-join jobs, which has never been considered in previous work.

Distributed algorithms. Distributed interval join evaluation was studied in [14]. The goal is to compute joins between 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. [5] implement the partitioning algorithm of [16] 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 [18]. Then, we review the forward scan based algorithm from [3], which has been overlooked by previous work. Both methods take as input two interval collections R and S and compute all (r, s) pairs ($r \in R, s \in S$), which intersect. We denote by r.start (r.end) the starting (ending) point of an interval r.

3.1 Endpoint-Based Interval Join

EBI [18] is based on the internal-memory plane sweep technique of [20] and 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 $\langle$endpoint, type, id$\rangle$, 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 collection (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 in the L1 cache or even in the cache registers, LEBI greatly reduces main memory cache misses and hence, outperforms EBI even more.

3.2 Forward Scan based Plane Sweep

The experiments in [18] showed that LEBI outperforms not only EBI, but also the plane sweep algorithm of [1], 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 [3], 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.[3] Algorithm 2 illustrates the pseudo-code of this method, denoted by FS. First, both input collections 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 and S in order. For each

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

ALGORITHM 2: Forward Scan based Plane Sweep (FS)

```
   Input  : collections of intervals R and S
   Output : set J of all intersecting interval pairs (r, s) ∈ R × S
1  J ← ∅;
2  sort R and S by start endpoint;
3  r ← first interval in R;
4  s ← first interval in S;
5  while R and S not depleted do
6      if r.start < s.start then
7          s' ← s;
8          while s' ≠ null and r.end ≥ s'.start do
9              J ← J ⋃ {(r, s')};                    ▷ add result
10             s' ← next interval in S;              ▷ scan forward
11         r ← next interval in R;
12     else
13         r' ← r;
14         while r' ≠ null and s.end ≥ r'.start do
15             J ← J ⋃ {(r', s)};                    ▷ add result
16             r' ← next interval in R;              ▷ scan forward
17         s ← next interval in S;
18 return J
```

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 $r.\mathsf{start} \leq s'.\mathsf{start} \leq r.\mathsf{end}$ (internal while-loops on Lines 7–10 and 13–16). Excluding the cost of sorting R and S, FS conducts $|R|+|S|+|R \bowtie S|$ endpoint comparisons, in total. Specifically, each interval $r \in R$ (the case for S is symmetric) is compared to just one $s' \in S$ which does not intersect r in the loop at Lines 8–10.

4. OPTIMIZING FS

In this section, we propose two optimization techniques for FS that can greatly enhance its performance, making it competitive to LEBI [18]. Note that the cost of FS cannot be asymptotically reduced as $|R|+|S|$ endpoint comparisons is the unavoidable cost of advancing the sweep line. Still, it is possible to reduce the number of $|R \bowtie S|$ comparisons required to produce the join results.

4.1 Grouping

The intuition behind our first optimization is to group consecutively sweeped intervals from the same collection and produce join results for them in batch, avoiding redundant comparisons. We exemplify this idea using Figure 1, 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 $r_1.\mathsf{start} < s_2.\mathsf{start}$, the next interval where the sweep line stops is r_1. Algorithm 2 (Lines 7–10) then forward scans through the shaded area in Figure 1(a) from $s_2.\mathsf{start}$ until it reaches $s_4.\mathsf{start} > r_1.\mathsf{end}$, producing result pairs $\{(r_1, s_2), (r_1, s_3)\}$. The next stop of the sweep line is $r_2.\mathsf{start}$, since $r_2.\mathsf{start} < s_2.\mathsf{start}$. FS scans through the shaded area in Figure 1(b) producing results $\{(r_2, s_2), (r_2, s_3), (r_2, s_4)\}$. 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 $s_2.\mathsf{start}$ and $s_3.\mathsf{start}$ which were compared to both $r_1.\mathsf{end}$ and $r_2.\mathsf{end}$. However, since $r_2.\mathsf{end} > r_1.\mathsf{end}$ holds, $r_1.\mathsf{end} > s_2.\mathsf{start}$ automatically implies that $r_2.\mathsf{end} > s_2.\mathsf{start}$; therefore, pairs (r_1, s_2) and (r_2, s_2) could have been reported by comparing only $r_1.\mathsf{end}$ to $s_2.\mathsf{start}$. Hence, processing consecutively sweeped 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: FS with grouping (gFS)

```
   Input  : collections of intervals R and S
   Output : set J of all intersecting interval pairs (r, s) ∈ R × S
1  sort R and S by start endpoint;
2  r ← first interval in R;
3  s ← first interval in S;
4  J ← ∅;
5  while R and S not depleted do
6      if r.start < s.start then
7          G^R ← next group from R w.r.t. r, s;
8          sort G^R by end endpoint;
9          s' ← s;
10         foreach r_i ∈ G^R in order do
11             while s' ≠ null and s'.start ≤ r_i.end do
12                 foreach r_j ∈ G^R, j ≥ i do
13                     J ← J ⋃ {(r_j, s')};          ▷ update results
14                 s' ← next interval in S;          ▷ scan forward
15         r ← first interval in R after G^R;
16     else
17         G^S ← next group from S w.r.t. s, r;
18         sort G^S by end endpoint;
19         r' ← r;
20         foreach s_i ∈ G^S in order do
21             while r' ≠ null and r'.start ≤ s_i.end do
22                 foreach s_j ∈ G^S, j ≥ i do
23                     J ← J ⋃ {(r', s_j)};          ▷ update results
24                 r' ← next interval in R;          ▷ scan forward
25         s ← first interval in S after G^S;
26 return J
```

Algorithm 3 illustrates the pseudo-code of gFS, which enhances FS with the *grouping optimization*. Instead of processing one 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 $r.\mathsf{start} < s.\mathsf{start}$ (the other case is symmetric). gFS, starting from r, accesses all $r' \in R$ such that $r'.\mathsf{start} < s.\mathsf{start}$ (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 *forward scan* to 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 $s'.\mathsf{start} \leq r_i.\mathsf{end}$, then s' intersects not only with r_i but also with all intervals in G^R *after* r_i (due to the sorting of G^R by end). If $s'.\mathsf{start} > r_i.\mathsf{end}$, 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 $r_i \in G^R$.

Figures 1(c) and (d) exemplify gFS for intervals r_1 and r_2 grouped under G^R; as $r_1.\mathsf{end} < r_2.\mathsf{end}$, r_1 is considered first. When the shaded area in Figure 1(c) from $s_2.\mathsf{start}$ until $s_4.\mathsf{start}$ is scanned, gFS produces results that pair both r_1 and r_2 with covered intervals s_2 and s_3 from S, by comparing $s_2.\mathsf{start}$ and $s_3.\mathsf{start}$ only to $r_1.\mathsf{end}$. Intuitively, avoiding redundant endpoint comparisons corresponds to removing the overlap between the scanned areas of consecutive intervals (compare r_2's scanned area by gFS in Figure 1(d) to the area in Figure 1(b) by FS after removing the overlap with r_1's area).

Discussion and implementation details. The grouping technique of gFS differs from the buffering employed by LEBI. First, LEBI groups consecutive start endpoints in a sort order that includes 4 sets of endpoints, whereas in gFS there are only 2 sets of endpoints (i.e., only start endpoints of the two collections). As a result, the groups in gFS have higher probability 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

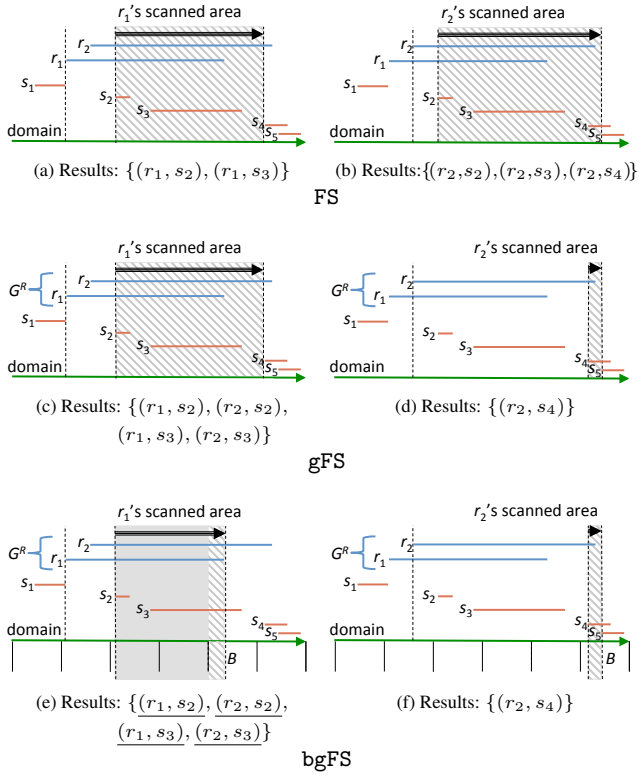

Figure 1: Scanned areas by FS, gFS and bgFS for intervals r_1 and r_2. Underlined result pairs are produced without any endpoint comparisons.

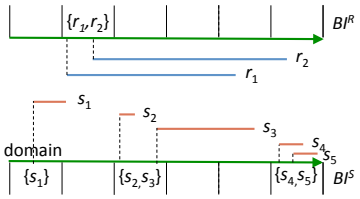

Figure 2: Domain tiles and BI^R, BI^S bucket indices for the intervals of Figure 1.

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 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 extends gFS to avoid more endpoint comparisons during the computation of join results. The idea is as follows. First, we split the domain into a predefined number of equally-sized disjoint tiles; all intervals from R (resp. S) that start within a particular tile are stored inside a dedicated bucket of the BI^R (resp. BI^S) *bucket index*. Figure 2 exemplifies the domain tiles and the bucket indices for the interval collections of

ALGORITHM 4: FS with grouping and bucket indexing (bgFS)

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

```
1  J ← ∅;
2  sort R and S by start endpoint;
3  build BI^R and BI^S;
4  r ← first interval in R;
5  s ← first interval in S;
6  while R and S not depleted do
7      if r.start < s.start then
8          G^R ← next group from R w.r.t. r, s;
9          sort G^R by end endpoint;
10         s' ← s;
11         foreach r_i ∈ G^R do
12             B ← bucket in BI^S: B.start ≤ r_i.end < B.end;
13             while s' is before B do                    ▷ no comparisons
14                 foreach r_j ∈ G^R, j ≥ i do
15                     J ← J ⋃ {(r_j, s')};               ▷ update results
16                 s' ← next interval in S;               ▷ scan forward
17             while s' ≠ null and s'.start ≤ r_i.end do
18                 foreach r_j ∈ G^R, j ≥ i do
19                     J ← J ⋃ {(r_j, s')};               ▷ update results
20                 s' ← next interval in S;               ▷ scan forward
21         r ← first interval in R after G^R;
22     else
23         G^S ← next group from S w.r.t. s, r;
24         sort G^S by end endpoint;
25         r' ← r;
26         foreach s_i ∈ G^S do
27             B ← bucket in BI^R: B.start ≤ s_i.end < B.end;
28             while r' is before B do                    ▷ no comparisons
29                 foreach s_j ∈ G^S, j ≥ i do
30                     J ← J ⋃ {(s_j, r')};               ▷ update results
31                 r' ← next interval in R;               ▷ scan forward
32             while r' ≠ null and r'.start ≤ s_i.end do
33                 foreach s_j ∈ G^S, j ≥ i do
34                     J ← J ⋃ {(s_j, r')};               ▷ update results
35                 r' ← next interval in R;               ▷ scan forward
36         s ← first interval in S after G^S;
37 return J
```

Figure 1.[4] With the bucket indices, the area scanned by gFS for an interval is entirely covered by a range of tiles. Consider Figures 1(c) and 1(e); r_1's scanned area lies inside three tiles which means that the involved intervals from S start between the BI^S bucket covering s_2.start and the BI^S bucket covering r_1.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 tiles/buckets, for such intervals s_i.start $\leq r_1$.end. Hence, we only need to conduct endpoint comparisons for the s_i intervals originating from the bucket that covers r_1.end. This distinction is graphically shown in Figures 1(e) and (f) where solid gray areas are used to directly produce join results with no endpoint comparisons. Observe that, for this example, all four join results produced when gFS performs a forward scan for r_1 are directly reported by bgFS.

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

PARADIGM 1: Hash-based Partitioning

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

1 $J \leftarrow \emptyset$;
2 **foreach** *interval* $r \in R$ **do** ▷ partition R
3 $v \leftarrow h(r)$; ▷ apply hash function
4 **add** r to partition R_v;
5 **foreach** *interval* $s \in S$ **do** ▷ partition S
6 $v \leftarrow h(s)$; ▷ apply hash function
7 **add** s to partition S_v;
8 **foreach** *partition* R_i *of* R **do**
9 **foreach** *partition* S_j *of* S **do**
10 $J \leftarrow J \bigcup \{R_i \bowtie S_j\}$; ▷ using LEBI, FS, gFS, bgFS
11 **return** J

PARADIGM 2: Domain-based Partitioning

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

1 $J \leftarrow \emptyset$;
2 **split** domain into k tiles;
3 **foreach** *interval* $r \in R$ **do** ▷ partition R
4 $t_{\mathsf{start}} \leftarrow$ domain tile covering $r.\mathsf{start}$;
5 $t_{\mathsf{end}} \leftarrow$ domain tile covering $r.\mathsf{end}$;
6 **add** r to partition R_{start};
7 **foreach** *tile* t_j *inside* $(t_{\mathsf{start}}, t_{\mathsf{end}}]$ **do**
8 **replicate** r to partition R_j;
9 **foreach** *interval* $s \in S$ **do** ▷ partition S
10 $t_{\mathsf{start}} \leftarrow$ domain tile covering $s.\mathsf{start}$;
11 $t_{\mathsf{end}} \leftarrow$ domain tile covering $s.\mathsf{end}$;
12 **add** s to partition S_{start};
13 **foreach** *tile* t_j *inside* $(t_{\mathsf{start}}, t_{\mathsf{end}}]$ **do**
14 **replicate** s to partition S_j;
15 **foreach** *domain tile* t_j **do**
16 $J \leftarrow J \bigcup \{R_j \bowtie S_j\}$; ▷ using LEBI, FS, gFS, bgFS
17 **return** J

Algorithm 4 illustrates the pseudo-code of bgFS which enhances gFS with *bucket indexing*. Essentially, bgFS operates similar to gFS. Their main difference lies in the forward scan for every interval inside the current group. Lines 12–20 implement the range query discussed in the previous paragraph. The algorithm first identifies bucket $B \in BI^S$ which covers $r_i.\mathsf{end}$. 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 13–16 without any endpoint comparison, while finally on Lines 17–20, the intervals of B are scanned and compared as in gFS.

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. In contrast, we store for each bucket a pointer to the last interval in it; this allows bgFS 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.

5. PARALLEL PROCESSING

We now shift our focus to the parallel execution of interval joins that benefits from the existence of multiple CPU cores in a system. We first revisit and critique the hash-based partitioning approach suggested in [18], and then, discuss our domain-based partitioning.

5.1 Hash-based Partitioning

In [18], Piatov et al. primarily focused on optimizing EBI for minimizing the memory access cost in modern hardware. However, the authors also described how EBI (and its lazy LEBI version) can be parallelized. In this spirit, a *hash-based* partitioning paradigm was proposed, described by Paradigm 1. The evaluation of the join involves two phases. First, the input collections are split into k disjoint partitions using a 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 to each other and hence, results are produced without the need of a duplicate elimination (i.e., merging) step.

In [18], 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 at Line 10 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 total number of 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 $2|R|+2|S|$ 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 $k(|R|+|S|)$ and $2k(|R|+|S|)$, respectively (i.e., an increase by a factor of k).

In addition, despite the even distribution of the load, the hash-based partitioning paradigm does not take full advantage of the available hardware. In order to fully take advantage of parallelism, each of the k^2 joins should be computed by a separate thread running on a dedicated processor (i.e., core). Hence, if there is a limited number n of CPU cores, we should set $k = \sqrt{n}$ to achieve this, i.e., the number of partitions is much smaller than the number of cores. In the next section, we present a *domain-based* partitioning paradigm, which creates n partitions for each input collection by splitting the intervals domain, being able to achieve a higher level of parallelism compared to the hash-based paradigm, independently of the underlying join algorithm.

5.2 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 2–14) splits the domain uniformly into k non-overlapping tiles; a partition R_j (resp. S_j) is created for each domain tile t_j. Let t_{start}, t_{end} denote the tiles that cover $r.\mathsf{start}$, $r.\mathsf{end}$ of an interval $r \in R$, respectively. Interval r is first assigned to partition R_{start} created for tile t_{start}. Then, r is *replicated* across tiles $t_{\mathsf{start}+1} \ldots t_{\mathsf{end}}$. The replicas of r carry a special flag (e.g., $\hat{r}$). During the second phase (Lines 15–16), the domain-based paradigm computes $R_j \bowtie S_j$ for every domain tile t_j, independently. To avoid producing duplicate results, a join result (r, s) is reported if *at least one* of the involved intervals is original (i.e., its replica flag is not set). 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 tile t_{j-1}, therefore (r, s) will be reported by another partition-join.

We illustrate the difference between the two paradigms using the intervals in Figure 1; without loss of generality, assume there are 4 CPU cores available to compute $R \bowtie S$. The hash-based paradigm will first create $\sqrt{4} = 2$ partitions for each input, i.e., $R_1 = \{r_1\}$,

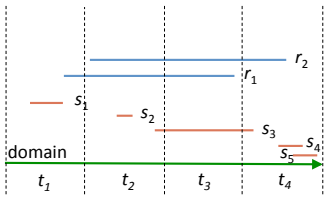

Figure 3: Domain-based partitioning of the intervals in Figure 1; the case of 4 available CPU cores.

$R_2 = \{r_2\}$ 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 tiles pictured in Figure 3, and then assign and replicate (if needed) the intervals into 4 partitions for each collection; $R_1 = \{r_1\}$, $R_2 = \{\hat{r}_1, r_2\}$, $R_3 = \{\hat{r}_1, \hat{r}_2\}$, $R_4 = \{\hat{r}_2\}$ 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 $r_i \in R$ (resp. $s_i \in S$) inside tile 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_2, s_4) but not (r_2, s_4) which will be found in $R_2 \bowtie S_2$.

Our domain-based partitioning paradigm achieves a higher level 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., [5, 16]), we avoid the production and elimination of duplicate join results. On the other hand, *long lived* intervals that span a large number of tiles and skewed distributions of start endpoints create joins of imbalanced costs. In what follows, we propose two orthogonal techniques that deal with load balancing.

5.2.1 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 tile 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 tile 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):

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

(ii) r starts inside a previous tile but ends inside t_j, i.e., $r.\mathsf{start} < t_j.\mathsf{start}$ and $r.\mathsf{end} < t_j.\mathsf{end}$, or

(iii) r starts inside a previous tile and ends after t_j, i.e., $r.\mathsf{start} < t_j.\mathsf{start}$ and $r.\mathsf{end} \geq t_j.\mathsf{end}$.

Note that in cases (ii) and (iii), r is assigned to partition R_j by replication (Lines 7–8 and 13–14 of Paradigm 2). We use $R_j^{(i)}$, $R_j^{(ii)}$, and $R_j^{(iii)}$ (resp. $S_j^{(i)}$, $S_j^{(ii)}$, and $S_j^{(iii)}$) 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:

- $R_j^{(i)} \bowtie S_j^{(i)}$ is evaluated as normal; i.e, as discussed in Sections 3 and 4.
- For $R_j^{(i)} \bowtie S_j^{(ii)}$ and $R_j^{(ii)} \bowtie S_j^{(i)}$, join algorithms only visit end endpoints in $S_j^{(ii)}$ and $R_j^{(ii)}$, respectively; $S_j^{(ii)}$ and $R_j^{(ii)}$ only contain replicated intervals from previous tiles which are properly flagged to precede all intervals starting inside t_j, and so, they form the sole group from $S_j^{(ii)}$ and $R_j^{(ii)}$ at gFS / bgFS.
- $R_j^{(i)} \bowtie S_j^{(iii)}$ and $R_j^{(iii)} \bowtie S_j^{(i)}$ reduce to cross-products, because replicas inside mini-partitions $S_j^{(iii)}$ and $R_j^{(iii)}$ span the entire tile t_j; hence, all interval pairs are directly output as results without any endpoint comparisons.
- $R_j^{(ii)} \bowtie S_j^{(ii)}$, $R_j^{(iii)} \bowtie S_j^{(ii)}$, $R_j^{(iii)} \bowtie S_j^{(ii)}$, $R_j^{(iii)} \bowtie S_j^{(iii)}$ are not executed at all as intervals from both inputs start in a previous tile, so the results of these mini-joins would be duplicates.

Given a fixed number n of available CPU cores, i.e., partitioning of the domain into $k = n$ tiles, our goal is to assign *each* of the $1 + 5 \cdot (k-1)$ in total mini-joins[5] to a core, in order to evenly distribute the load among all cores, or else to minimize the maximum load per core. This is a well known NP-hard problem, which we opt to solve using a classic $(4/3 - 1/3n)$-approximate algorithm [10] that has very good performance in practice. The algorithm greedily assigns to the CPU core with the currently least current 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 core 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 $R_j^{(i)} \bowtie S_j^{(i)}$, we *remove* bag b_p at the top of $\mathcal{Q}$ corresponding to core p with the least load, we *append* $R_j^{(i)} \bowtie S_j^{(i)}$ to b_p and *re-insert* the bag to $\mathcal{Q}$. This 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 core. The breakdown of each partition-join task into mini-joins that can be executed at different CPU cores greatly improves load balancing in the case where the original tasks have big cost differences.

5.2.2 Adaptive Partitioning

Our second *adaptive partitioning* technique for load balancing re-positions the endpoints of the $\{t_1, \ldots, t_k\}$ tiles, 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 cores, 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.[6] This problem can be solved exactly in PTIME with respect to the domain size, with the help of dynamic programming [12]; 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

[5]The only possible mini-join for the first tile is $R_j^{(i)} \bowtie S_j^{(i)}$, as it is not possible for it to contain any replicas.

[6]We assume that there is a function that can 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 tile, which holds in our case.

creating histograms in large domains that do not have quality guarantees but compute a good solution in practice within short time [19]. Note that, in practice, the overall execution time is dominated by the most expensive partition-join. Hence, given as input an initial set of tiles and partitions (more details in the next paragraph), we perform the following steps. First, the CPU core or equivalently the tile 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 tile 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 core with the highest load until no further moving of the load is possible.

The implementation of this heuristic raises two important challenges; (a) how we can quickly estimate the load on each of the $n = k$ available CPU cores and (b) what is the smallest unit of load (in other words, the smallest number of intervals) to be moved in between cores/tiles. 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., $\xi = c \cdot k$, where $c >> 1$. For each granule g, we count the number of intervals $H^R[g]$ and $H^S[g]$ from R and S respectively that start in g. We define every initial tile t_j as a set of consecutive c granules; in practice, this partitions the input collections into tiles of equal widths as our original framework. Further, we select a granule as the smallest unit (number of intervals) to be moved between tiles. The load on each core depends on the cost of the corresponding partition-join. This cost is optimized if we break it down into mini-joins, as described in Section 5.2.1, because numerous comparisons are saved. Empirically, we observed that the cost of the entire bundle of the 5 mini-joins that correspond to a tile t_j is dominated by the first mini-join, i.e., $R_j^{(i)} \bowtie S_j^{(i)}$, the cost of which can be estimated by $|R_j^{(i)}| \cdot |S_j^{(i)}|$. Hence, in order to calculate $|R_j^{(i)}|$ (resp. $|S_j^{(i)}|$), 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 tile t_j by moving granules to/from t_j, cardinalities $|R_j^{(i)}|$, $|S_j^{(i)}|$ and the join cost estimate for t_j can be incrementally updated very fast.

Finally, we implement the process of reducing the load of a tile t_j by moving consecutive granules located either exactly after t_j.start or exactly before t_j.end. Moving the endpoints of a tile does not involve any physical operations, since we only bookkeep the segments of the initial partitions that should be assigned to other partitions; this is possible since H^R (H^S) retains the exact number of intervals inside each moved granule.

Discussion. We can easily combine adaptive partitioning with dynamic scheduling as the two techniques improve different parts of Paradigm 2, i.e., its first and second phase, respectively.

6. EXPERIMENTAL ANALYSIS

We finally present our experimental analysis on interval joins under both single-threaded and parallel processing environments.

6.1 Setup

Our analysis was conducted on a machine with 128 GBs of RAM and a dual 10-core Intel(R) Xeon(R) CPU E5-2687W v3 clocked at 3.10 GHz running Linux; with hyper-threading, we were able to run up to 40 threads. All methods were implemented in C++, optimized by forcing loop-unrolling and compiled using `gcc` (v5.2.1) with flags `-O3`, `-mavx` and `-march=native`. For multi-threading, we used OpenMP. We imported the implementations of `EBI/LEBI` [18], `OIP` [7] and `DIP` [4], kindly provided by the authors of the corresponding papers, to our source code. The setup of our benchmark is similar to that of [18], i.e., every interval contains two 64-bit endpoint attributes while the workload accumulates the sum of an XOR between the `start` endpoints on every result pair. Note that all data (input collections, index structures etc.) reside in main memory. Regarding `bgFS` we set the number of buckets equal to 1000 on each test, after tuning. Finally, for parallel join evaluation, we assume a fixed number of n available CPU cores each running a single thread (as in [18]). Following the discussion in Section 5, both hash-based and domain-based paradigms fully employ the available cores by creating $\sqrt{n}$ and n partitions, respectively.

Datasets. We experimented with two real-world datasets (WEBKIT and BOOKS) and a number of synthetic ones. WEBKIT records the file history from 2001 to 2016 in the git repository of the Webkit project (https://webkit.org), at a granularity of milliseconds; valid times indicate the periods when a file did not change. BOOKS records the transactions in 2013 at Aarhus public libraries at a granularity of days (https://www.odaa.dk/); valid times indicate the periods when a book is lent out. Table 1 summarizes the characteristics of WEBKIT and BOOKS while Figure 4 shows their temporal distributions, i.e., a histogram summarizing the durations of the intervals and the number of open (i.e., valid) intervals at each timestamp; the latter is an indicator for the selectivity of an interval join. Note that the durations follow an exponential distribution. While the intervals may start at random domain points, there are also times in the domain where there is a burst in the concentration of intervals; we call these time points *peaks*. Based on this observation, for our synthetic datasets, we generate a fraction of intervals having uniformly distributed `start` endpoints, while the remaining ones are generated following a normal distribution around a number of random peaks, with a deviation equal to 10% of the domain. The duration of all generated intervals follow an exponential distribution. Table 2 summarizes the characteristics of the synthetic datasets. We generated the collections varying their cardinality, the domain size, the average interval duration as a fraction of the domain size, the ratio of distinct timestamps over the domain size, the number of involved peaks and the peak cardinality ratio (i.e., the percentage of intervals generated around peaks).

Tests. We ran two types of tests on the real-world datasets: (1) an interval join using a uniformly sampled subset of each dataset as the outer input R and the entire dataset as the inner S (ratio $|R|/|S|$ varies inside $\{0.25, 0.5, 0.75, 1\}$)[7], and (2) a parallel self-join (i.e., with $|R| = |S|$) varying the number of available CPU cores (and threads) from 4 to 36. Regarding the synthetic datasets, we perform a series of only non-self joins; on each test, we vary one of the parameters in Table 2 while fixing the rest to their default value. In addition, we also run a parallel non-self join, again varying the number of available CPU cores from 4 to 36. To assess the performance of the methods, we measure their total execution time which includes sorting, indexing and partitioning costs, and the total number of endpoint comparisons; for `FS`, `gFS`, `bgFS` this covers both advancing the sweep line and forward scanning, but for `LEBI` it only covers advancing the sweep line. Note that each of partitioning, sorting and indexing is fully parallelized; their costs are negligible compared to the cost of sweeping and scanning to produce the results, which dominates the overall execution time. Regarding the adaptive partitioning, we conducted a series of tests to define multiplicative factor c. To avoid significantly increasing

[7]We also experimented with disjoint subsets observing similar behavior; the results are omitted due to lack of space.

Table 1: Characteristics of real-world datasets

	WEBKIT	BOOKS
Cardinality	$2,347,346$	$2,312,602$
Domain duration (secs)	$461,829,284$	$31,507,200$
Shortest interval (secs)	1	1
Longest interval (secs)	$461,815,512$	$31,406,400$
Avg. interval duration (secs)	$33,206,300$	$2,201,320$
Distinct timestamps/endpoints	$174,471$	$5,330$

Table 2: Characteristics of synthetic datasets

	value range	default value
Cardinality	$1M, 5M, 10M, 50M, 100M$	$10M$
Domain size	$10K, 50K, 100K, 500K, 1M$	$100K$
Avg. interval duration ratio [%]	$0.1, 0.5, 1, 5, 10$	1
Distinct endpoints ratio [%]	$1, 5, 10, 50, 100$	100
Number of peaks	$1, 2, 3, 4, 5$	3
Peak cardinality ratio [%]	$0, 25, 50, 75, 100$	50

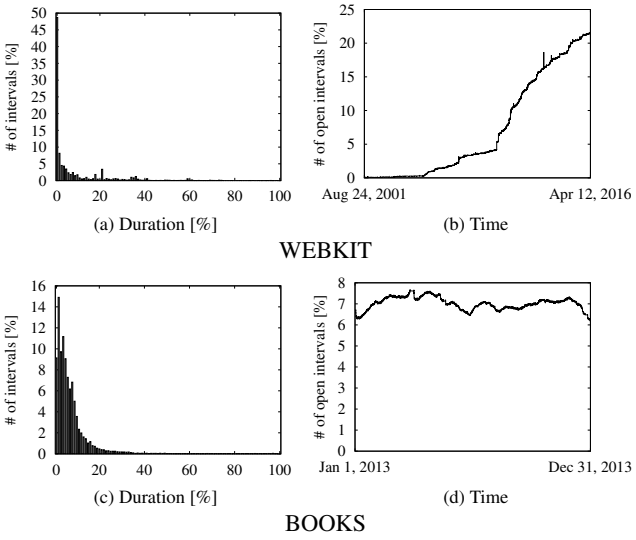

Figure 4: Temporal statistics of datasets

the partitioning cost, we ended up setting $c = 1000$ when the number of CPU cores is less than 16 and 4 or 9, and $c = 100$ otherwise. We indicate the activation of hyper-threading by 25_{HT} and 36_{HT}.

6.2 Optimizing FS

We first investigate the effectiveness of grouping and bucket indexing; these optimizations are orthogonal to the parallel processing of interval joins and so, we focus only on a single-threaded processing environment. Figure 5 reports the execution time and the ratio of conducted endpoint comparisons over the number of results for `FS`, `gFS`, `bgFS` on WEBKIT and BOOKS. The figures clearly demonstrate the effectiveness of grouping; `gFS` and `bgFS` both outperform `FS` on all tests; in fact, their advantage over `FS` becomes greater as we increase $|R|/|S|$, i.e., as the join becomes computationally harder and the result set larger. A larger $|R|/|S|$ implies in practice a small increase to the number of distinct endpoints in outer collection R; however, this is insignificant compared to the increase of collection's cardinality. As a result, both `gFS` and `bgFS` manage to create larger groups which allows them to further reduce the number of forward scans and avoid even more redundant endpoint comparisons; hence, the increasing performance gain over `FS`. Grouping is more beneficial in BOOKS due to a larger increase of the average group size compared to WEBKIT. Bucket indexing manages to further decrease the number of conducted comparisons, however, as Figures 5(a) and (c) show, `bgFS` cannot fully capitalize on this reduction. This is due to the overhead of producing the join result, which dominates the total execution time. Hence, `bgFS` outperforms `gFS` by a small margin on WEBKIT while on BOOKS the methods exhibit similar performance. For the rest of this analysis, `bgFS` is our default forward scan based plane sweep method.

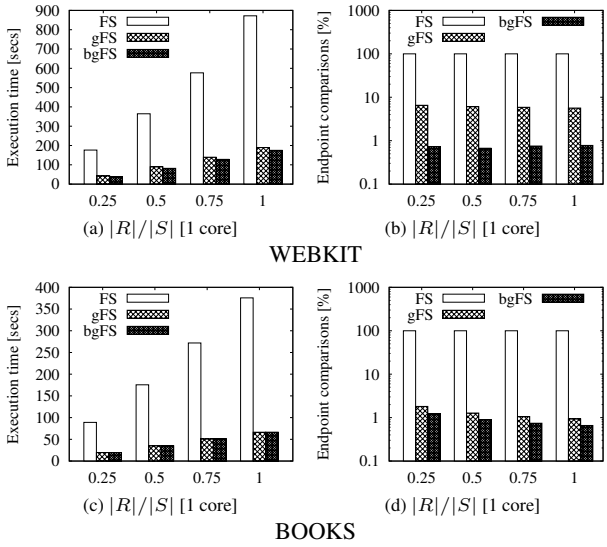

Figure 5: Optimizing `FS`

6.3 Comparisons: Single-threaded Processing

After optimizing `FS`, we compare `bgFS` against partition-based methods `DIP`, `OIP` and state-of-the-art plane sweep method `LEBI`. Figure 6 reports the execution times on ours WEBKIT, BOOKS and datasets INFECTIOUS, GREEND from [4]; INFECTIOUS, GREEND both contain very short intervals (of average duration 330K and 18M times smaller than their domain sizes, respectively). As expected, the execution time of all methods rises as we increase the $|R|/|S|$ ratio. At least one of `LEBI`, `bgFS` always outperforms their partition-based competitors; the results also align with in [18], where `LEBI` (and plane sweep based algorithms in general) were shown to outperform `OIP`. Finally, we also observe that `LEBI` outperforms `bgFS` by a small margin 10-20% in two out of the four datasets; recall that `LEBI` performs no endpoint comparisons to produce the results but for this purpose it relies on the gapless hash map. Nevertheless, `bgFS` stands as a decent competitor to `LEBI` in these two datasets, while it significantly outperforms `LEBI` on the other two.

6.4 Optimizing Domain-based Partitioning

Next, we study the impact of our optimization techniques for the domain-based partitioning paradigm. Due to lack of space, we only show the results for `bgFS` on WEBKIT; the same conclusions can be drawn from `bgFS` on BOOKS and from `LEBI` on both datasets. Besides the overall execution time of each join, we also measured the load balancing among the participating CPU cores. Let set $L = \{\ell_1 \dots \ell_n\}$ be the measured time spent by each of the available n cores; we define the *average idle time* as:

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

A high average idle time means that the cores 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 optimization denoted by *mj* (Section 5.2.1), the greedy scheduling technique denoted by *greedy* (Section 5.2.1), and adaptive partitioning denoted by *adaptive* (Section 5.2.2). We also use the term *atomic* to denote the assignment of each partition-join or

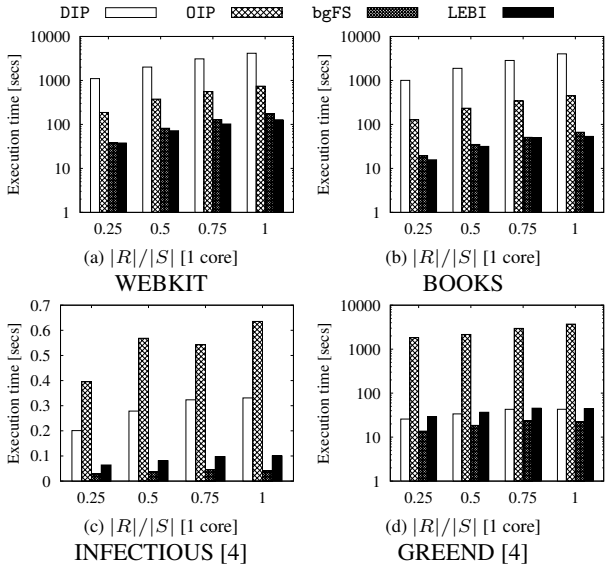

Figure 6: Comparisons for single-threaded processing

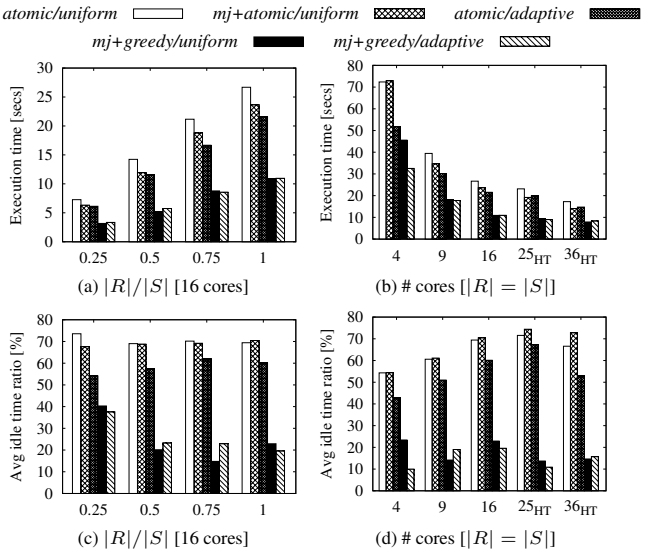

Figure 7: Optimizing the domain-based partitioning: bgFS on WEBKIT

the bundle of its corresponding 5 mini-joins to the same core, and *uniform* to denote the (non-adaptive) uniform initial partitioning of the domain. We tested the following setups:[8]

(1) *atomic/uniform* is the baseline domain-based partitioning of Section 5.2 with all optimizations deactivated;

(2) *mj+atomic/uniform* splits each partition-join of the baseline domain-based paradigm into 5 mini-joins which are all executed on the same CPU core;

(3) *atomic/adaptive* employs only adaptive partitioning;

(4) *mj+greedy/uniform* splits each partition-join of the baseline domain-based paradigm into 5 mini-joins which are greedily distributed to the available CPU cores;

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

Figures 7(a) and (b) report the total execution time of bgFS for each optimization combination (1)–(5) while Figures 7(c) and (d) report the ratio of the average idle time over the total execution time.

We observe the following. First, setups (2)–(5) all manage to enhance the parallel computation of the join. Their execution time is lower than the time of baseline *atomic/uniform*; an exception arises for *mj+atomic/uniform* under 4 available cores. 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 (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 core would be dedicated to its most expensive mini-join and the other mini-joins would be handled by less loaded CPU cores. Also, notice that the *mj* optimization is beneficial even when the 5 defined mini-joins are all executed on the same CPU core (i.e., *mj+atomic/uniform*). 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 seems to have a smaller impact compared to the other two optimizations. Among the setups that do not employ the *greedy* scheduling, *atomic/adaptive* ranks first (both in terms of the execution time the average idle time ratio) but when activated on top of the *mj+greedy/uniform* setup, adaptive partitioning enhances the join evaluation when the number of cores is low, e.g., 4 or 9; notice how faster is the *mj+greedy/adaptive* setup compared to *mj+greedy/uniform* in case of 4 available CPU cores.

Overall, (i) the *mj* optimization greatly reduces the cost of a partition join and adds flexibility in load balancing, (ii) the *mj+greedy/uniform* and *mj+greedy/adaptive* schemes peform very well in terms of load balancing, by reducing the average idle time of any core to less than 20% of the total execution time in almost all cases ($|R|/|S| = 0.25$ is the only exception). To take full advantage of all proposed optimizations, we setup the domain-based paradigm as *mj+greedy/adaptive* for the remaining of this analysis.

6.5 Comparisons: Parallel Processing

In this section, we first compare the domain-based partitioning against the hash-based proposed in [18]; this study is independent of the join algorithm we may use to compute partition- or mini-joins. Further, we compare our proposed implementation of FS with all optimizations (i.e., bgFS) to the state-of-the-art (as shown in Section 6.3) LEBI for parallel computation of interval joins.

Hence, we implemented the domain-based and the hash-based paradigms of Section 5 coupled with both LEBI and our best method bgFS, denoted by h-LEBI, d-LEBI and h-bgFS, d-bgFS; note that the *mj+greedy/adaptive* optimizations evaluated in the previous section are all activated on the LEBI powered implementation of the domain-based paradigm. As discussed in Section 5.1, [18] sorts each input collection prior to partitioning. We experimented also with a variant of the hash-based paradigm, which does not perform this pre-sorting step and proved to be always faster. Thus, for the rest of this subsection we run our variant of the hash-based partitioning. Figures 8(a)–(d) and Figures 9(a)–(d) report on this first comparison for both WEBKIT and BOOKS datasets; we show the speedup achieved by each parallel paradigm over the single-core evaluation (either with LEBI or bgFS) and the number of conducted endpoint comparisons. To better prove our points, we also include a third paradigm denoted as *theoretical* which exhibits a linear to the number of available cores, speedup and reduction of the conducted comparisons. We observe that our domain-based paradigm

[8]Based on our assumption in Section 6.1, *greedy/uniform* or *greedy/adaptive* setups are meaningless since the number of partitions equals the number of available CPU cores.

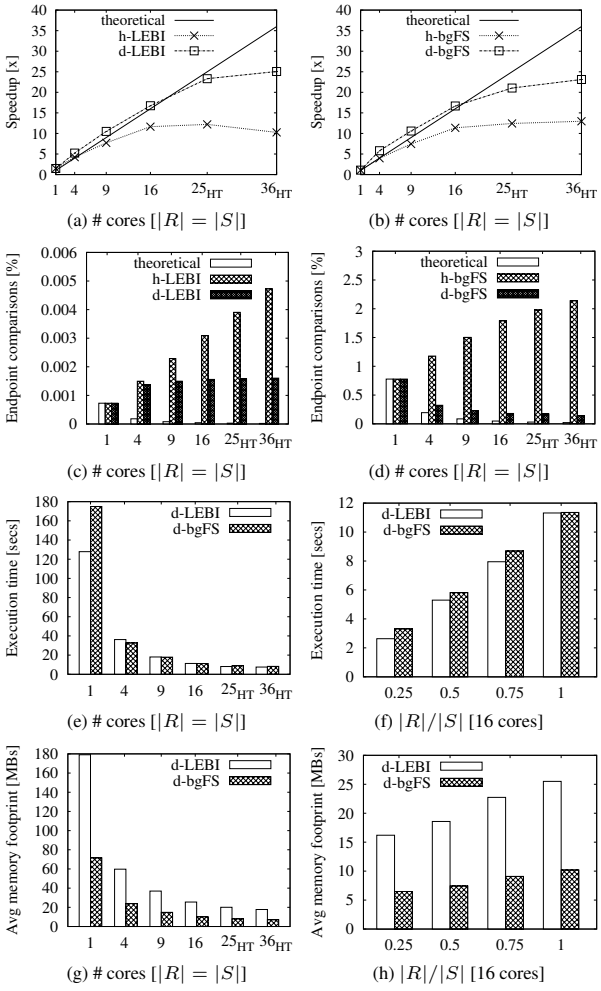

Figure 8: Comparisons for parallel processing on WEBKIT

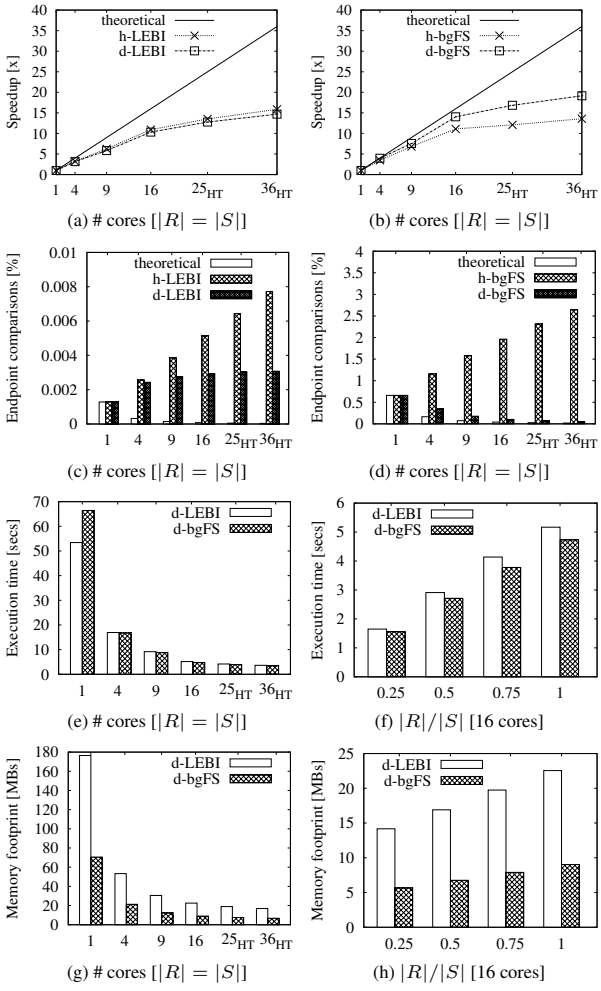

Figure 9: Comparisons for parallel processing on BOOKS

is more efficient than the hash-based, being able to achieve a greater speedup; in fact, on WEBKIT up to 16 cores, `d-LEBI` and `d-bgFS` take full advantage of parallelism, having the theoretically best possible speedup (for more than 16 cores, both paradigms are affected by hyper-threading, although they still scale well for 25 cores).

The benefits of the domain-based parallel processing are more apparent on WEBKIT; in fact, `d-LEBI` and `h-LEBI` exhibit the same speedup on BOOKS while `d-bgFS` always beats `h-bgFS` on either dataset. In practice, the interval joins on WEBKIT are more expensive than on BOOKS, producing a larger number of results as we can deduce from the open intervals distribution in Figure 4(b), (d). In this spirit, WEBKIT benefits more from the ability of the domain-based paradigm to significantly reduce the number of conducted endpoint comparisons as shown in Figures 8(c), (d) and Figures 9(c), (d). In fact, these figures experimentally prove our analysis at the end of Section 5.1 that employing hash-based paradigm increases the total number of comparisons compared even to a single-threaded algorithm, as the number of available CPU cores goes up.

In addition, note that the number of comparisons for `d-LEBI` increases with the number of cores in contrast to `d-bgFS`. This is an expected behavior. Recall that `LEBI` and hence, also `d-LEBI`, compare the endpoint of intervals only to advance the sweep line; as the number of partitions increases, so does the number of replicated intervals which reflects on the total number of endpoint comparisons. Partially, this is also the case for `d-bgFS`. However, the total number of endpoint comparisons on `FS`-based methods is dominated by the comparisons performed to produce the join results; the domain-based paradigm allows `d-bgFS` to significantly prune redundant comparisons during this step.

Figures 8(e), 8(f) (WEBKIT) and Figures 9(e), 9(f) (BOOKS) compare the relative performance of `bgFS` and `LEBI` under multi-core processing environments. Compared to Figure 6, we observe that `LEBI` is no longer the most efficient method; in some cases, `d-bgFS` actually outperforms `d-LEBI`, but generally speaking their execution time is very similar. The reason for the relative improvement of `bgFS` over `LEBI` in parallel processing is the breakdown of partition-joins into mini-joins, which greatly reduces the total cost for comparisons by `d-bgFS` at each partition (while it does not affect the cost of `d-LEBI` that much). Besides the fact that `d-bgFS` is much simpler compared to `d-LEBI` in terms of the required data structures, it also has much lower space requirements per core, as shown in Figures 8(g), 8(h) (WEBKIT) and Figures 9(g), 9(h) (BOOKS). This is due to the fact that `LEBI`/`d-LEBI` has to build an endpoint index for each collection (partition), which contains double the amount of entries present in the input.

Finally, we report on the synthetic datasets; due to lack of space we do not show comparisons between hash-based and domain-based partitioning; the results are similar to the case of the real-world datasets showing the advantage of our domain-based paradigm. Figure 10 compares `d-bgFS` with `d-LEBI` as a function of input cardinality, average interval duration, domain size, number of distinct endpoints, number of peaks, and peak cardinality ratio. In general,

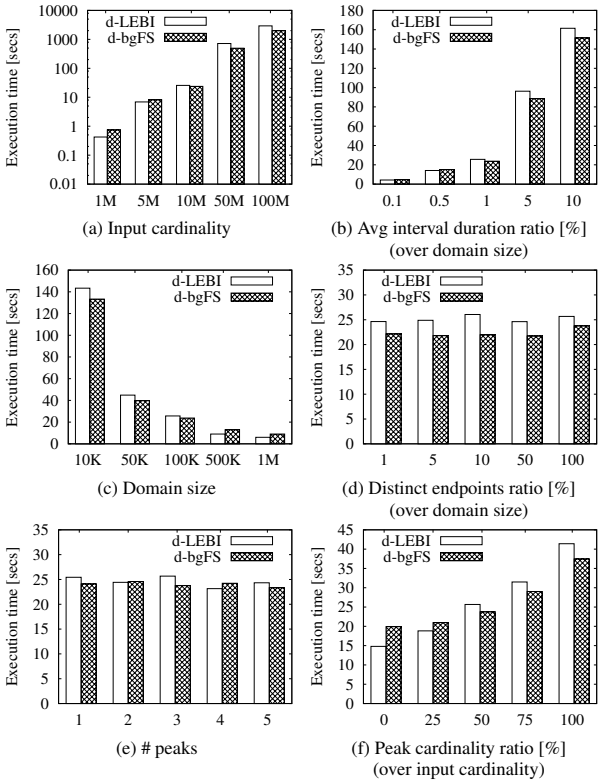

Figure 10: Parallel processing on synthetic data [16 cores]

d-LEBI and d-bgFS exhibit similar performance; however, there exist setups which benefit more either of the methods.

Consider first Figures 10(a), (b) and (f). The execution time of both methods increases with the input cardinality, the average interval duration and peak cardinality ratio as the result set becomes larger and the join more expensive. Nevertheless, we observe that d-bgFS scales better in all three cases. With the domain size and the number of distinct endpoints fixed, bgFS creates increasingly larger groups (the number of groups remains practically the same) which benefits the grouping optimization. In contrast, the size of d-LEBI's buffer remains fixed (to fit in the L1 cache). Figure 10(c) shows that d-LEBI scales better with the increase of the domain size. With the ratio of distinct endpoints fixed to its default value (100%), bgFS creates an increasingly larger number of groups which, however, contain fewer records; so, the effect of the grouping deteriorates. Last in Figure 10(d), we observe that increasing the number of distinct endpoints under a fixed domain size has very little effect in the time of both methods. In practice, d-bgFS creates increasingly more groups of fewer intervals but compared to varying the domain size, these groups are large enough to enhance grouping. Very large groups do not offer additional advantage to d-bgFS due to the increase of L1 cache misses when scanning them to produce results.

7. CONCLUSIONS

In this paper, we studied FS, a simple and efficient algorithm for interval joins based on plane sweep that does not rely on any special data structures. We proposed two novel optimizations for FS that greatly reduce the number of incurred comparisons making it competitive to the state-of-the-art. We also studied the problem of parallel interval joins, by proposing a domain-based partitioning framework. We showed that each partition-join can be broken down to five independent mini-joins, out of which, the four that involve replicated intervals have significantly lower cost than a standard interval join problem. We showed how to assign the threads that implement the mini-joins to a (smaller) number of CPU cores and how to improve the domain partitioning by the help of statistics. Our experimental evaluation suggests that (i) our optimized version of FS is significantly faster than the simple algorithm, and (ii) our domain-based partitioning framework for parallel joins significantly outperforms the hash-based framework suggested in [18] and scales well with the number of CPU cores.

8. REFERENCES

[1] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In *VLDB*, 1998.

[2] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *VLDB J.*, 5(4):264–275, 1996.

[3] T. Brinkhoff, H. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. In *SIGMOD*, 1993.

[4] F. Cafagna and M. H. Böhlen. Disjoint interval partitioning. *VLDB J.*, 26(3):447–466, 2017.

[5] B. Chawda, H. Gupta, S. Negi, T. A. Faruquie, L. V. Subramaniam, and M. K. Mohania. Processing interval joins on map-reduce. In *EDBT*, 2014.

[6] R. Cheng, S. Singh, S. Prabhakar, R. Shah, J. S. Vitter, and Y. Xia. Efficient join processing over uncertain data. In *CIKM*, 2006.

[7] A. Dignös, M. H. Böhlen, and J. Gamper. Overlap interval partition join. In *SIGMOD*, 2014.

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

[9] D. Gao, C. S. Jensen, R. T. Snodgrass, and M. D. Soo. Join operations in temporal databases. *VLDB J.*, 14(1):2–29, 2005.

[10] R. L. Graham. Bounds for multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17:416–429, 1969.

[11] H. Gunadhi and A. Segev. Query processing algorithms for temporal intersection joins. In *ICDE*, 1991.

[12] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *VLDB*, 1998.

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

[14] H. Kriegel, P. Kunath, M. Pfeifle, and M. Renz. Distributed intersection join of complex interval sequences. In *DASFAA*, 2005.

[15] H. Kriegel, M. Pötke, and T. Seidl. Managing intervals efficiently in object-relational databases. In *VLDB*, 2000.

[16] T. Y. C. Leung and R. R. Muntz. Temporal query processing and optimization in multiprocessor database machines. In *VLDB*, 1992.

[17] B. Moon, I. F. V. López, and V. Immanuel. Efficient algorithms for large-scale temporal aggregation. *TKDE*, 15(3):744–759, 2003.

[18] D. Piatov, S. Helmer, and A. Dignös. An interval join optimized for modern hardware. In *ICDE*, 2016.

[19] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. In *SIGMOD*, 1996.

[20] F. P. Preparata and M. I. Shamos. *Computational Geometry - An Introduction*. Texts and Monographs in Computer Science. Springer, 1985.

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

[22] I. Sitzmann and P. J. Stuckey. Improving temporal joins using histograms. In *DEXA*, 2000.

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

[24] D. Zhang, V. J. Tsotras, and B. Seeger. Efficient temporal join processing using indices. In *ICDE*, 2002.