A Forward Scan based Plane Sweep Algorithm for Parallel Interval Joins

Panagiotis Bouros  
Aarhus University, Denmark  
pbour@cs.au.dk

Nikos Mamoulis  
The University of Hong Kong, China  
nikos@cs.hku.hk

March 2, 2017

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 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 proposed two novel optimized versions of FS that greatly reduce its cost, making it competitive to the state-of-the-art single-threaded algorithm. 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 a smaller number of 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 \(\text{start}, \text{end} \in \mathbb{N}\), such that \(\text{start} \leq \text{end}\), we can define an interval \(i = [\text{start}, \text{end}]\) as the subset of \(\mathbb{N}\), which includes all integers \(x\), such that \(\text{start} \leq x \leq \text{end}\).\(^1\) Assume two collections of intervals \(R\) and \(S\). The interval join \(R \bowtie S\) is defined by all pairs of intervals \(r \in R\) and \(s \in S\) 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 operators in temporal databases [8]. 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.

The interval join also finds application in other domains. For example, in multi-dimensional spaces, an object can be represented as a set of intervals in a space-filling curve. The intervals correspond to the

\(^1\)Note that the intervals that we will deal with in this paper are closed. However, our techniques and discussions are applicable with minor changes for generic intervals where the begin and end sides can be either open or closed.
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 [13]. 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 [5].

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 [18] has not been considered as a competitive approach in most of the previous work.\(^2\) In fact, only a recent paper [16] 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., from \(R\)), the corresponding interval is removed from the respective active set (e.g., \(A^R\)).

The work of [16] 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 3, we present two novel optimizations for FS, which greatly reduce the number of comparisons during the join computation. In particular, after using our optimizations, FS can produce multiple join tuples in batch at the cost of a single comparison. Our optimized version of FS achieves competitive performance to the state-of-the-art algorithm (EBI [16]), without using any special hardware optimizations.

Our second contribution (Section 4) is an optimized framework for processing plane sweep based algorithms in parallel. We first show that the hash-based partitioning framework suggested in [16] 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. Duplicate avoidance was not studied in previously proposed parallel and distributed interval join frameworks that use domain-based partitioning [4, 15]. 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 [16]. 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 the state-of-the-art interval join algorithm.

The rest of the paper is organized as follows. Section 2 reviews in detail the state-of-the-art interval join algorithm [16] and the basic version of FS [3]. In Section 3, we propose two novel optimizations for FS that greatly reduce the computational cost of the algorithm in practice. Section 4 presents our domain-based partitioning framework for parallel interval joins. Section 5 includes our experimental evaluation which demonstrates the effect of our optimizations to FS and the efficiency of our parallel interval join framework. In Section 6, we review related work on interval joins and Section 7 concludes the paper.

\(^2\)We believe the main reason is that previous work mostly focused on centralized, hard-disk based evaluation, which becomes less relevant nowadays with the reduction in the cost of computational resources and the wide availability of parallel and distributed platforms and models.
**Algorithm 1:** Endpoint-Based Interval (EBI) Join

**Input:** collections of intervals $R$ and $S$

**Output:** set $J$ of all intersecting interval pairs $(r, s) \in R \times S$

**Variables:** endpoint indices $EI^R$ and $EI^S$, active interval sets $A^R$ and $A^S$

1. $J \leftarrow \emptyset, A^R \leftarrow \emptyset, A^S \leftarrow \emptyset$
2. build $EI^R$ and $EI^S$;
3. sort $EI^R$ and $EI^S$ first by endpoint then by type;
4. $e^R \leftarrow$ first index tuple in $EI^R$;
5. $e^S \leftarrow$ 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. $r \leftarrow$ interval in $R$ with identifier $e^R$.id;
         10. add $r$ to $A^R$;
         11. foreach $s \in A^S$ do
             12. $J \leftarrow J \cup \{(r, s)\}$;
             13. \hspace{1cm} update results
      14. else
         15. remove $r$ from $A^R$;
         16. $e^R \leftarrow$ next index tuple in $EI^R$;
   17. else
      18. if $e^S$.type = START then
         19. $s \leftarrow$ interval in $S$ with identifier $e^S$.id;
         20. add $s$ to $A^S$;
         21. foreach $r \in A^R$ do
             22. $J \leftarrow J \cup \{(r, s)\}$;
             23. \hspace{1cm} update results
      24. else
         25. remove $s$ from $A^S$;
         26. $e^S \leftarrow$ next index tuple in $EI^S$;
27. return $J$

2 Preliminaries

In this section, we present the necessary background for our analysis. First, we detail the state-of-art method for interval joins [16]. Then, we review the forward scan based plane sweep algorithm from [3], which has been overlooked by previous studies. 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 $r$.

2.1 Endpoint-Based Interval Join

Piatov et al. [16] proposed the Endpoint-Based Interval (EBI) join algorithm, which is based on the internal-memory plane sweep technique of [18] 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 (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 with their start endpoints have been accessed but not the tuples with their end. 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 $e^R$).

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 $ET^R$ and $ET^S$, 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 11–23 (scan). 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.

### 2.2 Forward Scan based Plane Sweep

The experimental analysis in [16] showed that LEBI outperforms not only EBI, but also the plane sweep algorithm of [1], which directly scans the input collections 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.

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 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 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 \(r.\text{end}\), i.e., all \(s' \in S\) with \(r.\text{start} \leq s'.\text{start} \leq r.\text{end}\) (internal while-loops on Lines 7–10 and 13–16). Excluding the cost of sorting collections \(R\) 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.

3 Optimizing FS

In this section, we propose two novel optimization techniques for FS that can greatly enhance its performance, making it competitive to the state-of-the-art LEBI algorithm [16]. Note that the cost of FS cannot be asymptotically reduced since \(|R|+|S|\) 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.

3.1 Grouping

The intuition behind our first optimization 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 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 \(r_1.\text{start} < s_2.\text{start}\), 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 \(s_2.\text{start}\) until it reaches \(s_4.\text{start} > r_1.\text{end}\), producing result pairs \(\{(r_1, s_2), (r_1, s_3)\}\). The next stop of the sweep line is \(r_2.\text{start}\), since \(r_2.\text{start} < s_2.\text{start}\). 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 \(s_2.\text{start}\) and \(s_3.\text{start}\) which were compared to both \(r_1.\text{end}\) and \(r_2.\text{end}\). However, since \(r_2.\text{end} > r_1.\text{end}\) holds, \(r_1.\text{end} > s_2.\text{start}\) automatically implies that \(r_2.\text{end} > s_2.\text{start}\); therefore, pairs \((r_1, s_2)\) and \((r_2, s_2)\) could have been reported by comparing only \(r_1.\text{end}\) to \(s_2.\text{start}\). 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 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 \(r.\text{start} < s.\text{start}\) (the other case is symmetric). gFS, starting from \(r\), accesses all \(r' \in R\) such that \(r'.\text{start} < s.\text{start}\) (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 \(s'.\text{start} \leq r_i.\text{end}\), 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 \(s'.\text{start} > r_i.\text{end}\), 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 \(r_1.\text{end} < r_2.\text{end}\), \(r_1\) is considered first. When the shaded area in Figure 1(c) from \(s_2.\text{start}\) until \(s_4.\text{start}\) 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 \(s_2.\text{start}\) and \(s_3.\text{start}\) only to \(r_1.\text{end}\). 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

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

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 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).
Algorithm 3: gFS with grouping (gFS)

**Input**: collections of intervals $R$ and $S$

**Output**: set $J$ of all intersecting interval pairs $(r, s) \in R \times S$

1. sort $R$ and $S$ by start endpoint;
2. $r \leftarrow$ first interval in $R$;
3. $s \leftarrow$ first interval in $S$;
4. $J \leftarrow \emptyset$;
5. while $R$ and $S$ not depleted do
   6. if $r$.start < $s$.start then
      7. $G^R \leftarrow$ next group from $R$ w.r.t. $r, s$;
      8. sort $G^R$ by end endpoint;
      9. $s' \leftarrow s$;
      10. foreach $r_i \in G^R$ in order do
          11. while $s' \neq \text{null}$ and $s'$.start $\leq r_i$.end do
              12. $J \leftarrow J \cup \{(r_i, s')\}$;
              13. $s' \leftarrow$ next interval in $S$;
          ▷ update results
          ▷ 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 $r' \neq \text{null}$ and $r'$.start $\leq s_i$.end do
              21. $J \leftarrow J \cup \{(r', s_i)\}$;
              22. $r' \leftarrow$ next interval in $R$;
          ▷ update results
          ▷ scan forward
      23. $s \leftarrow$ first interval in $S$ after $G^S$;
   24. return $J$

---

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

3.2 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
ALGORITHM 4: FS with grouping and bucket indexing (bgFS)

\[ J \leftarrow \emptyset; \]
\[ \text{sort } R \text{ and } S \text{ by start endpoint;} \]
\[ \text{build } BI^R \text{ and } BI^S; \]
\[ r \leftarrow \text{first interval in } R; \]
\[ s \leftarrow \text{first interval in } S; \]
\[ \text{while } R \text{ and } S \text{ not depleted do} \]
\[ \quad \text{if } r.\text{start} < s.\text{start then} \]
\[ \quad \quad G^R \leftarrow \text{next group from } R \text{ w.r.t. } r, s; \]
\[ \quad \quad \text{sort } G^R \text{ by end endpoint;} \]
\[ \quad \quad s' \leftarrow s; \]
\[ \quad \quad \text{foreach } r_i \in G^R \text{ do} \]
\[ \quad \quad \quad B \leftarrow \text{bucket in } BI^S; B.\text{start} \leq r_i.\text{end} < B.\text{end}; \]
\[ \quad \quad \quad \text{while } s' \text{ is before } B \text{ do} \]
\[ \quad \quad \quad \quad J \leftarrow J \cup \{(r_j, s')\}; \]
\[ \quad \quad \quad \quad s' \leftarrow \text{next interval in } S; \]
\[ \quad \quad \quad \text{update results} \]
\[ \quad \quad \text{scan forward} \]
\[ \quad \quad \text{else} \]
\[ \quad \quad G^S \leftarrow \text{next group from } S \text{ w.r.t. } s, r; \]
\[ \quad \quad \text{sort } G^S \text{ by end endpoint;} \]
\[ \quad \quad r' \leftarrow r; \]
\[ \quad \quad \text{foreach } s_i \in G^S \text{ do} \]
\[ \quad \quad \quad B \leftarrow \text{bucket in } BI^R; B.\text{start} \leq s_i.\text{end} < B.\text{end}; \]
\[ \quad \quad \quad \text{while } r' \text{ is before } B \text{ do} \]
\[ \quad \quad \quad \quad J \leftarrow J \cup \{(s_j, r')\}; \]
\[ \quad \quad \quad \quad r' \leftarrow \text{next interval in } R; \]
\[ \quad \quad \quad \text{update results} \]
\[ \quad \quad \quad \text{scan forward} \]
\[ \quad \quad \text{return } J \]

the interval collections of 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 \(r_1\).start. In this spirit, area scanning resembles a range query over the bucket

\(^4\)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.
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 \( s_i \text{start} \leq r_1 \text{end} \). Hence, we only need to conduct endpoint comparisons for the \( s_i \) intervals originating from the bucket that covers \( r_1 \text{end} \). 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 \( \text{gFS} \) performs a forward scan for \( r_1 \) are directly reported by \( \text{bgFS} \).

Algorithm 4 illustrates the pseudo-code of \( \text{bgFS} \) which enhances \( \text{gFS} \) with bucket indexing. At an abstract level, \( \text{bgFS} \) operates similar to \( \text{gFS} \). The difference between the two methods 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 \( r_1 \text{end} \). Then, it iterates through the \( s_i \in S \) 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 \( \text{gFS} \).

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 \( \text{bgFS} \) 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.

4 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 [16], and then, discuss our domain-based partitioning.

4.1 Hash-based Partitioning

In [16], 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 [16], 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 11 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 \( \text{FS} \) and \( \text{EBI} \) 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.
4.2 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 2–13) 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 \( t_{\text{start}}, t_{\text{end}} \) denote the tiles that cover \( r_{\text{start}}, r_{\text{end}} \) of an interval \( r \in R \), respectively. Interval \( r \) is first assigned to partition \( R_{\text{start}} \) created for tile \( t_{\text{start}} \). Then, \( r \) is replicated across tiles \( t_{\text{start}+1} \ldots t_{\text{end}} \). The replicas of \( r \) carry a special flag (e.g., \( \hat{r} \)). During the second phase (Lines 15–16), the domain-based paradigm computes \( R_j \join S_j \) for every domain tile \( t_j \), independently. In order to avoid producing duplicate join results, a join result \((r, s)\) is reported if \textit{at least one} of the involved intervals in the result 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., \( R_1 = \{r_1\}, R_2 = \{r_2\} \) 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 \( R_2 \bowtie S_2 \). On the other hand, 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; \( R_1 = \{r_1\}, R_2 = \{r_1, r_2\}, R_3 = \{r_1, r_2\}, R_4 = \{r_2\} \) for \( R \) and \( S_1 = \{s_1, s_3, s_5\}, S_2 = \{s_2, s_3, s_4\}, S_3 = \{s_3\}, S_4 = \{s_3, s_4, s_5\} \) for \( S \), where \( \hat{r}_j \) (resp. \( \hat{s}_j \)) denotes the replica of an interval \( r_j \in R \) (resp. \( s_j \in S \)) 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,
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. $t_{\text{start}} \leftarrow$ domain tile covering $r.\text{start}$;
   5. $t_{\text{end}} \leftarrow$ domain tile covering $r.\text{end}$;
   6. add $r$ to partition $R_{t_{\text{start}}}$;
   7. foreach tile $t_j$ inside $[t_{\text{start}}, t_{\text{end}}]$ do
      ▷ replicate $r$ to partition $R_j$;
9. foreach interval $s \in S$ do ▷ partition $S$
10. $t_{\text{start}} \leftarrow$ domain tile covering $s.\text{start}$;
11. $t_{\text{end}} \leftarrow$ domain tile covering $s.\text{end}$;
12. add $s$ to partition $S_{t_{\text{start}}}$;
13. foreach tile $t_j$ inside $[t_{\text{start}}, t_{\text{end}}]$ do
      ▷ replicate $s$ to partition $S_j$;
15. foreach domain tile $t_j$ do
16. $J \leftarrow J \cup \{R_j \triangleright S_j\}$; ▷ using LEBI, FS, gFS, bgFS
17. return $J$

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., [4, 15]), 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.

4.2.1 Mini-joins and Greedy Scheduling

Our first optimization of Paradigm 2 is based on decomposing the partition-join $R_j \triangleright S_j$ 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 $t_j.\text{start}$ and $t_j.\text{end}$ 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., $t_j.\text{start} \leq r.\text{start} < t_j.\text{end}$,

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

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

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 $R_j \triangleright S_j$ down to 9 distinct mini-joins, out of which only 5 need to be evaluated (and the evaluation is simplified for 4 out of these 5 mini-join tasks). Specifically:

- $R_j^{(i)} \triangleright S_j^{(i)}$ is evaluated as normal; i.e, as discussed in Sections 2 and 3.

- For $R_j^{(i)} \triangleright S_j^{(ii)}$ and $R_j^{(ii)} \triangleright S_j^{(ii)}$, the 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 hence, they form for instance the sole group from $S_j^{(ii)}$ and $R_j^{(ii)}$ under gFS and bgFS.
\begin{itemize}
\item $R_j^{(i)} \Join S_j^{(iii)}$ and $R_j^{(iii)} \Join S_j^{(i)}$ 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.
\item $R_j^{(i)} \Join S_j^{(i)}$, $R_j^{(iii)} \Join S_j^{(i)}$, $R_j^{(iii)} \Join S_j^{(ii)}$, $R_j^{(iii)} \Join S_j^{(iii)}$ are not executed at all, because intervals from both inputs start in a previous tile, so the results of these mini-joins would be duplicates.
\end{itemize}

Given a fixed number $n$ of available CPU cores, i.e., $k = n$ partitions of the domains into 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 [9] 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 cardinality 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 $\ell_p$, 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 $Q$ based on their load. Finally, we examine all mini-joins in descending order of their estimated cost. For each mini-join say $R_j^{(i)} \Join S_j^{(i)}$, we remove the bag $b_p$ at the top of $Q$ which corresponds to the CPU core $p$ with the lowest load, we append $R_j^{(i)} \Join S_j^{(i)}$ to $b_p$ and re-insert the bag to the queue. 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 (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.

### 4.2.2 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 [11]; 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 [17]. 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 $\ell_j$) 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 $t_j$’s start or retracting $t_j$’s 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 these 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 $\xi$ of granules with $\xi$ being a large multiple of $k$, i.e., $\xi = c \cdot k$, where $c >> 1$. For each granule $g$, we count the number of

\(^5\)The only possible mini-join for the first tile is $R_j^{(i)} \Join S_j^{(i)}$, 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.
using gcc to 40 threads. All methods were implemented in C++, optimized by forcing loop-unrolling and compiled E5-2687W v3 clocked at 3.10GHz running Linux; hyper-threading was activated so, we were able to run up
Our analysis was conducted on a machine with 128 GBs of RAM and a dual 10-core Intel(R) Xeon(R) CPU
5.1 Setup

We next present our experimental analysis on interval joins both under a single-core and a multi-core pro-

improve different parts of Paradigm 2, i.e., its first and second phase, respectively.

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.

5 Experimental Analysis

We next present our experimental analysis on interval joins both under a single-core and a multi-core processing environment.

5.1 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 v3clocked at 3.10GHz running Linux; hyper-threading was activated so, 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 API v4.0. We imported their implementation of EBI/LEBI provided by [16] to our source code. The setup of our benchmark is similar to [16], i.e., every interval contains two 64-bit endpoint attributes while the workload consisted of accumulating 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, which both the hash-based and the domain-based paradigm fully employ, i.e., by creating \(\sqrt{n}\) as discussed in Section 4.1 and \(n\) partitions, respectively (a similar setup to [16]).

Datasets. We experimented with two real-world datasets (WEBKIT and BOOKS) and synthetic ones. WEBKIT records the files history at the git repository of the Webkit project (https://webkit.org) from 2001 to 2016, at a granularity of milliseconds; valid times indicate the periods when a file did not change. BOOKS records the transactions on Aarhus public libraries (https://www.odaa.dk/) in 2013 at a granularity of days; 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, that is 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 durations of all generated intervals follow an exponential distribution. Table 2 summarizes the characteristics of our synthetic datasets. We generated the collections varying their cardinality, the domain density via the average interval duration, the number of involved peaks
Table 1: Characteristics of datasets

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

Table 2: Characteristics of the synthetic datasets

<table>
<thead>
<tr>
<th></th>
<th>value range</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cardinality</td>
<td>100K, 500K, 1M, 5M, 10M</td>
<td>1M</td>
</tr>
<tr>
<td>Domain size</td>
<td>100K</td>
<td>100K</td>
</tr>
<tr>
<td>Avg. interval duration</td>
<td>100, 500, 1K, 5K, 10K</td>
<td>1K</td>
</tr>
<tr>
<td>Number of peaks</td>
<td>1, 2, 3, 4, 5</td>
<td>3</td>
</tr>
<tr>
<td>Peak cardinality ratio [%]</td>
<td>0, 25, 50, 75, 100</td>
<td>50</td>
</tr>
</tbody>
</table>

Figure 4: Temporal statistics of datasets

and the peak cardinality ratio (i.e., the percentage of intervals generated around peaks). The domain size is fixed to 100K units.

Tests. We ran two types of tests on the real-world datasets: (1) an interval join using a subset of each dataset as the outer input \( R \) and the entire dataset as the inner \( S \) (ratio \( |R|/|S| \) varies inside \( \{0.25, 0.5, 0.75, 1\} \))\(^7\).

\(^7\)We also experimented with disjoint subset of datasets observing similar behavior.
and (2) a parallel self-join (i.e., with \(|R| = |S|\)) varying the number of available CPU cores 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, while for LEBI only 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 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 in the figures by \(25_{HT}\) and \(36_{HT}\).

### 5.2 Optimizing FS

We first investigate the effectiveness of the grouping and bucket indexing optimizations; these techniques are orthogonal to the parallel processing of interval joins and so, we focus only on a single-core 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 both 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. Processing intervals in groups enables the methods to avoid redundant endpoint comparisons; as a result, \(gFS\) reduces the number of endpoint comparisons in \(FS\) by more than 90\%. We also observe that
bucket indexing manages to further decrease the number of conducted comparisons; especially in WEBKIT, bgFS’s comparisons ratio is an order of magnitude lower than that of gFS. Unfortunately, as Figures 5(a) and (c) show, bgFS cannot fully capitalize on this reduction. The reason is 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. Overall, none of our tests showed bucket indexing to have a negative impact on top of gFS and thus, for the rest of this analysis, we consider bgFS as our default forward scan based plane sweep method.

5.3 Optimizing Domain-based Partitioning

Next, we study the impact of our optimization techniques for the domain-based partitioning paradigm. 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 \( L = \{\ell_1 \ldots \ell_n\} \) 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 4.2.1), the greedy scheduling technique denoted by \( greedy \) (Section 4.2.1), and adaptive partitioning denoted by \( adaptive \) (Section 4.2.2). We also use the term \( atomic \) to denote the assignment of each partition-join or 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:

- (1) \( atomic/uniform \) is the baseline domain-based partitioning of Section 4.2 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 6(a) and (b) report the total execution time of bgFS for each optimization combination (1)–(5) while Figures 6(c) and (d) report the ratio of the average idle time over the total execution time.

We make the following observations. 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 sizes (2 of them are simple 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 due to the fact that by breaking down a partition-join into 5 mini-joins, we greatly reduce 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,  

8Based on our assumption in Section 5.1, \( greedy/uniform \) or \( greedy/adaptive \) setups are meaningless since the number of partitions equals the number of available CPU cores.
Figure 6: Optimizing the domain-based partitioning paradigm: bgFS on WEBKIT
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 perform 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.

### 5.4 Comparisons

In this section, we first compare the domain-based partitioning under a multi-core processing environment against the hash-based proposed in [16]; this study is independent of the join algorithm we may use to compute partition- or mini-joins. In addition, we compare our proposed implementation of FS with all optimizations (i.e., bgFS) against the state-of-the-art LEBI algorithm under both single-core and multi-core processing environments.

Hence, we implemented the domain-based and the hash-based paradigms of Section 4 coupled with both state-of-the-art 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 4.1, [16] sort 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 7(a)–(d) and Figures 8(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 partitioning paradigm is more efficient than the hash-based paradigm, 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 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; on the other hand, 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 distribution of the open intervals in Figure 4(b) and (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 7(c), (d) and Figures 8(c), (d). In fact, these figures experimentally prove our analysis at the end of Section 4.1 that employing hash-based partitioning actually 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 7(e), 7(f) (WEBKIT) and Figures 8(e), 8(f) (BOOKS) compare the relative performance of bgFS and LEBI in both single-core and multi-core processing environments. d-LEBI outperforms d-bgFS on a single-core evaluation by 10-20%; in this case, both methods reduce to LEBI and bgFS, respectively. This is expected as LEBI performs no endpoint comparisons to produce the results and relies on a sophisticated data structure (the gapless hash map) to minimize the cost of maintaining and accessing the active sets. Nevertheless, d-bgFS stands as a decent competitor. On the other hand, as we turn to parallel join evaluation, d-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 d-bgFS...
Figure 7: Comparisons on WEBKIT
Figure 8: Comparisons on BOOKS
over d-LEBI in multi-core join 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, as shown in Figures 7(g), 7(h) (WEBKIT) and Figures 8(g), 8(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 in brief on the synthetic datasets; regarding the comparisons between hash-based and the domain-based partitioning; the results are similar to the case of the real-world datasets showing the advantage of our domain-based paradigm. Figure 9 compares d-bgFS with d-LEBI as a function of input cardinality, average interval duration, number of peaks, and peak cardinality ratio. We again observe that d-bgFS is as efficient as d-LEBI on parallel processing of interval joins.

### 6 Related Work

In this section, we briefly 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 [10, 19] 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. [7] propose interval join algorithms, which operate on two RI-trees
Zhang et al. [22] 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 [10, 19]), proposing an extension of the multi-version B-tree [2].

**Partitioning-based algorithms.** A partitioning-based approach for interval joins was proposed in [21]. The domain is split to 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 [20]. A more sophisticated partitioning approach, called Overlap Interval Partitioning (OIP) Join [6], 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 [7] and sort-merge join. These results are consistent with the comparative study of [8], which shows that partitioning-based methods are superior to nested loops and merge join approaches.

**Methods based on plane sweep.** The Endpoint-Based Interval (EBI) Join (reviewed in detail in Section 2.1) is the most recent approach and we consider it to be the state-of-the-art. EBI and its lazy version (LEBI) were shown to significantly outperform the best partitioning-based approach [6] and to also be superior to another plane sweep implementation [1]. An approach similar to EBI is used in SAP HANA [12]. To our knowledge, no previous work was compared to FS [3].

**Parallel algorithms.** A domain-based partitioning strategy, similar to that described in Section 4.2, for interval joins on multi-processor machines was proposed in [15]. Each partition is assigned to a processor and intervals replicated to the partitions they overlap, in order to ensure that join results can be produced independently at each processor. However, a merge phase with duplicate elimination is required because the same join result can be produced by different processors. Our parallel join processing approach (Section 4) also applies a domain-based partitioning but does not produce duplicates. In addition, our breakdown to mini-joins has never been proposed in previous work.

**Distributed algorithms.** A distributed interval join method is proposed in [13]. The goal is to compute joins between sets of intervals, which are located at different clients. The clients send statistics about the distribution of the local data to the server, which merges them to form global statistics. These statistics are sent back to the clients, based on which the latter compute and send more refined statistics. In turn, these statistics allow for a coarse-level approximate join at the server’s side; exact results are refined by on-demand communication with the clients. Chawda et al. [4] implement the partitioning algorithm of [15] 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. Our focus, on the other hand, is the CPU-efficient evaluation of interval joins in a shared-memory environment.

### 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 [16] and scales well with the number of CPU cores.
References


