

HCIJ: Hardware-Conscious Interval Joins for Modern CPUs

Ioannis Reppas
EURECOM
Biot, France
ioannis.reppas@eurecom.fr

Raja Appuswamy
EURECOM
Biot, France
raja.appuswamy@eurecom.fr

Abstract

Interval join is a widely used operation in temporal and spatial databases. While many algorithms have been proposed for efficiently evaluating interval joins, limited attention has been given to understanding their microarchitectural characteristics, or their ability to effectively exploit various aspects of modern CPUs, like SIMD vectorization, NUMA awareness, and thread scheduling, to enhance performance. In this work, we bridge this gap by making the following contributions: (i) We present the first microarchitectural analysis of a state-of-the-art interval join algorithm to show that it significantly underutilizes modern CPUs. We perform an in-depth root cause analysis to identify various bottlenecks that lead to such underutilization. (ii) We present HCIJ—Hardware-Conscious Interval Join, an interval join algorithm for modern, multisocket, multicore CPUs that overcomes various bottlenecks by being hardware conscious in its design for data partitioning (to balance utilization across cores), SIMD vectorization (to avoid data dependencies), handling NUMA topologies (explicit memory management), and data caching. (iii) We perform a thorough experimental evaluation to demonstrate that HCIJ is portable across multiple CPU architectures, and can outperform the state-of-the-art solutions $1.2\times - 3.2\times$ times while maintaining a smaller memory footprint.

Keywords

Interval Joins, Temporal Databases, Hardware-Conscious Query Processing

1 Introduction

Interval data processing is a specialized field that focuses on managing and querying data defined by one-dimensional ranges, such as time periods or spatial extents. It is particularly valuable in temporal and spatial databases, where the need to analyze overlapping or contained intervals is common. The increasing interest in this field is highlighted by the introduction of the overlaps predicate in the SQL:2011 standard [30]. The interval join, which is the operator we concentrate on, is one of the most used join operations in temporal databases [22] and is used in other domains too, like uncertain data management [14]. Consider the two temporal relations *Employee Contracts* and *Project Duration* in Figure 1. *Employee Contracts* records the contract period of employees using half-open intervals [Start, End). *Project Duration* records the time periods in which a project takes place in a company. The interval join between the two relations can give us the time periods in which each employee worked on a specific project.

Interval joins can be useful in a variety of real-world applications. For example, delivery companies might join a truck’s travel time with periods of heavy traffic to explain a late package, or hospitals might join the time a patient was given medicine with

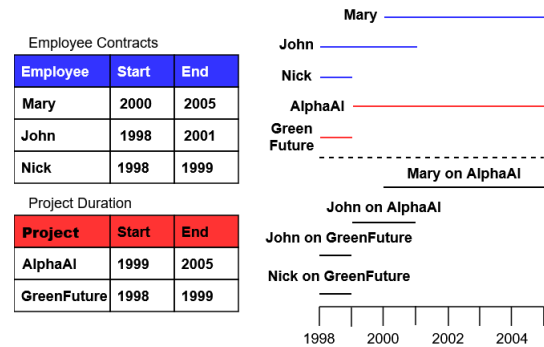


Figure 1: An example of an Interval Join.

the time their heart rate was high to check for side effects. Thousands of similar use-cases can be identified in modern big-data systems. Fast interval join computation is pivotal, as millions or even billions of records need to be compared to each other. Inefficient implementations cause applications to run for hours or even days, thus making it impossible to get answers quickly and wasting computer power.

Modern server hardware provides parallelism at various levels, including instruction-level parallelism and SIMD data parallelism in a processing core, task-level parallelism across cores in a multicore CPU, and multiprocessor parallelism on modern multisocket CPUs. Thus, a huge body of literature focuses on maximizing parallelism across various levels using both microarchitectural analysis studies [1, 29, 46, 47, 50], and macro-level scheduling studies, for relational database operations [6, 21, 26, 32]. However, the complexity of an interval join operation compared to selections, projections or even equality joins makes it more difficult for a developer to tune them to work on par with the underlying hardware. Previous work on studying their alignment to modern CPU characteristics is limited only to cache efficiency [40]. Additionally, recent state-of-the-art techniques for interval joins are based on plane-sweep strategies [2, 8–10, 40] (also common in spatial data management [3]), which are difficult to parallelize. To our knowledge, there is no work to systematically address these challenges and: (i) characterize interval joins at the microarchitectural level to understand bottlenecks, or (ii) holistically investigate the impact of key hardware aspects (like multi-core execution, SIMD vectorization, non-uniform memory access (NUMA), etc) to fully utilize modern CPUs. In this paper, we bridge this gap by making the following contributions:

- We present the first top-down microarchitectural analysis of interval joins. We characterize the stalls causing poor CPU utilization in state-of-the-art algorithms, identifying severe bottlenecks in vectorization, load balancing, and NUMA-awareness.
- We introduce Hardware-Conscious Interval Join (HCIJ), built around a novel data structure: the Forward Scan Map (FSMap). FSMap acts as an $O(1)$ cost oracle that

enables precise, output-aware load balancing, overcoming the inaccurate size-based estimates of the state-of-the-art. We further optimize HCIJ with an explicit SIMD forward scan, cache-bound grouping, and replication-based NUMA awareness.

- Through an extensive experimental evaluation on real-world datasets, we demonstrate that HCIJ outperforms state-of-the-art algorithms across multiple CPU architectures (x64, ARM, RISC-V) while achieving an order of magnitude smaller memory footprint.

2 Background and Motivation

In this section, we provide a short description about related work (Sec. 2.1) and how the state-of-the-art algorithms for interval joins work (Sec. 2.2). We then analyze their behavior on single and multicore environments to identify how well they exploit aspects of the underlying hardware (Sec. 2.3).

2.1 Previous work on Interval Joins

We classify proposed solutions for interval joins based on their functionality and describe the most relevant ones.

Different types of interval joins Many different problems of joining interval relations have been studied in the literature. For example, early solutions focused on defining the problem of interval joins in addition to non-interval value matching, and proposed simple techniques based on nested-loop or merge joins [24, 44]. Other operations for joining temporal relations based on different interval conditions, such as joins based on Allen relations [2, 39] have also been described. Recent works have also studied the problem of interval anti-joins [12, 43] and the joining of multiple interval relations [27]. In this work, we focus on the most common type of interval join that matches intervals from two relations, based solely on their overlap, as described in Figure 1.

Streaming Interval Joins Interval joins have also been studied in the field of streaming data processing. Several general-purpose works for streaming join operations, like the handshake join [49], can be used for streaming interval joins. Additionally, the operation of Online Interval Join [37, 53] has been defined in literature, studying the case of interval joins solely on streaming data. This operation, however, is fundamentally different to the offline version on static data that we discuss in this work, as data relations are treated as streams that arrive through time and are not fully saved in the memory or disk.

Index-based algorithms Prior research has also investigated index-based approaches for computing interval joins, both using traditional indexes, like RI-trees [20], B-trees [4, 19, 52] and spatial indexes [33], or using specialized indexes like Timeline-Index [28], HINT [16, 17] and RD-index [13]. While interval-specific index structures were demonstrated to provide better performance than generic indexing techniques, a major problem with all indexing approaches is that they require the pre-existence of the index for the fast computation of interval joins; for large interval joins, constructing indexes for an unindexed input – as well as due to the increased amount of random accesses for the index nodes traversal – has been shown to take much longer than non-index-based join execution [15]. Hence, indexed and unindexed methods are studied separately in interval-join literature as they target different use cases [7, 35]. In this work, we focus on unindexed interval joins.

Algorithm 1: optFS

Input: Relations R and S

Output: 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 while  $r \neq \text{null}$  and  $s \neq \text{null}$  do
5   if  $r.\text{start} < s.\text{start}$  then
6      $s' \leftarrow s$  ;
7     output all  $(r, s')$  until  $s'.\text{start} \geq r.\text{end}$  ;
8      $r \leftarrow$  next interval in  $R$  ;
9   else
10     $r' \leftarrow r$  ;
11    output all  $(r', s)$  until  $r'.\text{start} \geq s.\text{end}$  ;
12     $s \leftarrow$  next interval in  $S$  ;
```

Partitioning and plane sweep algorithms In the past few years, several algorithms have been proposed to accelerate unindexed interval joins. Notably, Overlap Interval Partitioning [18] (OIP) and Disjoint Interval Partitioning [12] (DIP) are partitioning strategies that outperformed both indexed and unindexed competitors. OIP creates partitions from equally-sized granules, while DIP partitions relations in a way that intervals that exist in the same partition do not overlap to each other. More recently though, the plane sweep algorithms of LEBI [40], optFS [9, 10] and different variations of them [19] were proposed to perform fast interval join processing, outperforming any previous algorithm on unindexed inputs, like OIP and DIP. Between the two, optFS (a forward-scan-based algorithm) was proved to be superior in terms of memory usage as it does not use any external data structures. We dedicate next Section to briefly describe how both work.

2.2 Plane-sweep based Interval Joins

Plane-sweep based approaches are widely considered the most efficient way to compute unindexed interval joins in modern literature [9–11, 19, 39, 40]. Considering that an interval is saved in the main memory as a tuple of two endpoints (start and end), as shown in Figure 1, and that both relations are pre-sorted, a plane sweep algorithm will scan both input relations based on their sorting condition. For each start endpoint accessed, a certain operation will be performed to report the result tuples that overlap to each other.

Algorithms 1 and 2 illustrate the pseudo-code of the two different plane-sweep approaches in modern literature, named optFS and LEBI, respectively. The difference between the two algorithms lies on the way that they report result tuples. optFS chooses to *forward scan* the future tuples according to their sorting order by start endpoint, based on a condition. On the other hand, LEBI keeps all active intervals in an external data structure, which prioritizes quick sequential scans.

Both algorithms work similarly if we also consider optimizations that can be applied on them. More specifically, both strategies use techniques like loop unrolling [38] and columnar layout [48] to accelerate performance. As optFS faces the challenge of performing many comparisons while reporting the results, it also splits the time domain into equally sized bucket indexes (Figure 4) to reduce comparisons. The most effective optimization

Algorithm 2: LEBI

Input: Relations R and S , Active Sets A^R and A^S
Output: all intersecting interval pairs $(r, s) \in R \times S$

```

1 Create Endpoint Indexes  $EI^R$  and  $EI^S$  ;
2 Sort  $EI^R, EI^S$  by endpoint and type ;
3  $e^R \leftarrow$  first endpoint in  $EI^R$  ;
4  $s^S \leftarrow$  first endpoint in  $EI^S$  ;
5 while  $e^R \neq \text{null}$  and  $e^S \neq \text{null}$  do
6   if  $e^R < e^S$  then
7     if  $e^R$  is of type start then
8       Add  $e^R.id$  to  $A^R$  ;
9       output  $(e^R.id, e'^S.id)$  where  $e'^S \in A^S$  ;
10    else
11      Remove  $e^R.id$  from  $A^R$  ;
12     $e^R \leftarrow$  next endpoint in  $EI^R$  ;
13  else
14    if  $e^S$  is of type start then
15      Add  $e^S.id$  to  $A^S$  ;
16      output  $(e'^R.id, e^S.id)$  where  $e'^R \in A^R$  ;
17    else
18      Remove  $e^S.id$  from  $A^S$  ;
19     $e^S \leftarrow$  next endpoint in  $EI^S$  ;

```

of both techniques is called *grouping*; consecutive elements in the plane sweep phase (either intervals or endpoints) get grouped and results are reported together either by forward scanning future intervals, or by reading the active sets.

2.3 Interval Joins through the Looking Glass

In this section, we perform a microarchitectural analysis on state-of-the-art interval join algorithms. Our goal is to identify gaps between the achieved performance, and the maximum performance levels that we can reach on modern CPUs.

2.3.1 CPU Microarchitecture. Modern processor pipelines are conceptually divided into a Front-End (FE) and Back-End (BE). The FE fetches and decodes program instructions into micro-operations (μOps), which are queued for execution by the BE. Before execution, all necessary data operands must be available, which may require fetching them from memory. Each μOp is assigned to an execution unit based on its type (e.g., arithmetic operations, memory access, etc). To improve performance, CPUs have the ability to use multiple independent execution units at the same time. Additionally, to reduce delays from slow DRAM accesses, both the FE and BE use on-chip and off-chip caches (L1/L2/L3 instruction/data caches). Once a μOp completes execution, it retires, meaning its results are committed to CPU registers or memory. In most microarchitectures, the pipeline's Front-End can allocate, and Back-End can retire up to four μOps per clock cycle. In recent designs, like the Intel's Golden Cove microarchitecture, the number of instructions that can be retired in each cycle can reach the number of six. Most μOps move through the pipeline and retire successfully. However, if a μOp cannot complete immediately, it may introduce stalls in the pipeline. Intel's Top-Down Analysis Methodology [51] categorizes stalls into three main types: a) Front-end stalls, b) Speculation stalls and c) Back-end stalls.

Front-End Stalls If the Back-End (BE) is ready to process a micro-operation (μOp) during a cycle but the Front-End (FE) is unable to supply it in time, the delay is categorized as a Front-End (FE) stall. These stalls are frequently caused by instruction cache misses which result from complex codebases with large instruction footprints. This issue is common in disk-based and OLTP databases [1, 47, 50].

Speculation Stalls To boost instruction throughput, modern processors rely on speculative execution. When a program contains conditional logic, such as if-else statements, these are compiled into branch instructions that direct the control flow based on certain conditions. For the processor to execute a branch, it must first determine the outcome of the condition. Rather than pausing until this result is known, the Front-End (FE) uses a Branch Predictor, which applies an algorithm to anticipate the likely outcome and begin fetching the corresponding instruction path. If the prediction is accurate, execution proceeds smoothly. However, if the prediction is incorrect, the pipeline must be flushed and the correct instruction stream loaded, causing delays. These delays, resulting from incorrect branch predictions, are referred to as speculation stalls.

Back-End Stalls When the Front-End (FE) has a μOp ready but the Back-End (BE) is not prepared to handle it, the processor experiences a BE stall. BE stalls can be further divided into memory stalls and core stalls. As noted earlier, a μOp can only be executed when its data operands are available. Memory stalls occur when the processor must wait for these operands to be fetched from cache or memory. On the other hand, core stalls arise from suboptimal use of the execution units, often due to resource contention. For example, if multiple μOps are tied to a limited number of division units, the queues for those units may fill up, preventing the scheduler from assigning additional μOps until some space is freed.

2.3.2 Microarchitectural analysis. To perform a microarchitectural analysis, we use a server equipped with 256GB of RAM, dual Intel(R) Xeon(R) Gold 5317 CPUs, with a base frequency of 3.00GHz, running Ubuntu 22.04. We analyze the implementation of the two state-of-the-art strategies for interval joins; optFS [10] and LEBI [40], provided to us by their authors. Detailed information about the hardware and software configuration is provided in Section 4.

We consider three real-world datasets of large interval lengths. The main properties of them are summarized in Table 1. The *Dig* dataset contains intervals from Chicago's Dig ticket notification system between the years 2018–2023. The *Requests* dataset records NYC311 Requests between the years 2010–2023. The *Emergency* dataset contains intervals on emergency department visits for Influenza-like Illness (ILI) from NYC OpenData. We generate the largest possible, evenly sized relations per dataset by uniformly distributing intervals. The input sizes are 6.5M (*Dig*), 18.5M (*Requests*), and 40M (*Emergency*). Since the intervals are exceptionally wide, all three queries yield massive output cardinalities – 1.47875×10^{12} for *Dig*, 2.738×10^{12} for *Requests*, and 1.088×10^{15} for *Emergency* –, making the post-sort join phase computationally heavy. Their selectivities ($|R \bowtie S|/|R \times S|$) are 0.035, 0.008, and 0.68, respectively.

We use the Intel VTune profiler for our analysis. Table 2 shows the microarchitectural analysis results (including stall types) for optFS and LEBI. We make several observations. First, the percentage of retired instructions is high across all datasets. Our CPU is able to retire up to 4 instructions per cycle (IPC). The

Table 1: Characteristics of Real-World Datasets

Dataset	Dig	Requests	Emergency
Cardinality	13,045,430	37,242,961	82,021,561
Domain Size	183,497,400	32,100,969,600	87,177,600
Shortest Interval	9,000	1	172,800
Longest Interval	79,867,800	31,878,244,918	87,177,600
Average Interval	3,142,491	1,863,312	29,449,118

results indicate that both plane-sweep algorithms are able to achieve an IPC between 2 and 3.2, which is something encouraging, considering that OLAP and OLTP database workloads are known to achieve IPC of less than 2, with the CPU being stalled most of the time [1, 46, 47, 50].

Second, to understand how we can further improve performance, we now focus on the stalls. We can observe that, in all cases, the high number of retired instructions is accompanied by a large percentage of core stalls. As previously explained, core stalls are a result of congestion for execution resources at the processor's units. Recall that modern CPUs have separate units for scalar execution, that work with one data item at a time, and vector instructions that provide SIMD execution [25]. Here is where we observed the first major difference between optFS and LEBI. Additional experiments and code analysis revealed that many retiring instructions in the code corresponding to forward scans (Lines 6 and 9 in Algorithm 1) in optFS were of the scalar type, suggesting that the compiler was not able to apply automatic vectorization to many parts of the optFS code. In contrast, LEBI was able to retire more vectorized instructions than optFS (1.1 $\times$ more in Dig and Emergency and 4.1 $\times$ more in Requests, measured with sde64 profiler). Surprisingly, despite this difference in vectorization, there is not a big difference in performance between the two techniques and LEBI, despite using in theory more operations due to external data structures, retires roughly the same instructions in total. We experimented with various compiler optimization flags, but were unsuccessful in enabling full auto-vectorization. On further examination, we found this to be due to the following reason. In Algorithm 1, the join computation is performed based on the end-condition check. This creates a dependency making it difficult for the compiler to apply auto-vectorization (as Requests dataset is executing many of these comparisons due to the distribution of its intervals in the time domain, its vector instructions decrease more compared to Dig and Emergency). In contrast, LEBI did not have this dependency and was able to vectorize code better. In the ideal scenario, we would like to combine the benefit of both approaches, that is exploit SIMD vectorization like LEBI (and potentially improve it even more), but without the additional overhead of external data structures, like optFS. While there has been a huge body of work focusing on vectorization of relational operations [5, 25, 42], to our knowledge, there has been no such work in the context of interval joins.

Third, memory stalls reveal significant disparities across datasets and algorithms: on Requests, memory stalls account for a substantial fraction of total cycles (15% for LEBI, 43.4% for optFS), whereas on Emergency and Dig they remain under 20% for optFS and 10% for LEBI. This discrepancy comes from an interaction between a characteristic existing at both algorithms—grouping. Both algorithms batch matches using groups of consecutive start intervals. During execution, a new group is brought into cache upon the first match and remains resident across

Table 2: Microarchitectural analysis

optFS	Dig	Requests	Emergency
Retiring	62.3 %	39.6 %	60 %
Core Stalls	24.7 %	14.2 %	20 %
Memory Stalls	9.7 %	43.4 %	18 %
Front-end Stalls	2 %	1.9 %	1.3 %
Speculation Stalls	1.3 %	0.8 %	0.7 %
IPC Rate	3.2	2	3.17
#Instructions (Billion)	1,000	8,800	570,000
LEBI	Dig	Requests	Emergency
Retiring	57 %	57.6 %	52 %
Core Stalls	31 %	20.9 %	8 %
Memory Stalls	9 %	15 %	10 %
Front-end Stalls	1.5 %	2.7 %	15 %
Speculation Stalls	1.5 %	3.8 %	15 %
IPC Rate	2.9	2.6	2.4
#Instructions (Billion)	900	7,800	580,000

the scan, whereas future/active set intervals are accessed once and are rarely cached beforehand. Thus, matching the first element of a group with a future interval incurs a memory stall, but subsequent group elements reuse that fetched interval directly from processor registers. Consequently, datasets with tiny group sizes—such as Requests (two on average)—amortize memory stalls poorly compared to Dig (82 on average). Additionally, extremely large group sizes—like in Emergency (40K on average)—exceed on-chip capacity. Because LEBI caps group size to the L1d cache limit, it avoids cache eviction issues on large groups. In contrast, optFS lacks this bound and large groups cause extra DRAM fetches.

Last, we focus on other stall types in Table 2. We see that, in most cases, speculation and front-end stalls are minimal. In database workloads, these categories are known to be the second dominant stall source after memory stalls due to complex branching and instruction footprint [1]. optFS and LEBI are optimized, in-memory implementations. Hence, they do not suffer from these stalls (excluding Emergency for LEBI, where they are still less than 20%).

2.3.3 Interval Joins on multicore CPUs. So far, we analyzed plane-sweep algorithms on one CPU core. However, a modern CPU consists of multiple cores, each with its own private L1/L2 caches. Hence, we now investigate how well they exploit the task-level parallelism of multicore CPUs.

Plane-sweep strategies are difficult to parallelize. Thus, in order to compute interval joins in parallel, the state-of-the-art paradigm uses a Domain-based Partitioning algorithm, first introduced in [31] and later optimized in [9, 10], which works independently of whether the single-core algorithm running underneath is a plane-sweep one. This algorithm splits the time domain on which the intervals exist into a set of non-overlapping partitions, such that one partition can be assigned to each available CPU thread. Intervals that belong in more than one partition get replicated. Each interval receives a category label between $\{A, B, C\}$, depending on whether the interval starts, ends or fully crosses a partition, respectively. Intervals that both start and end in the same partition receive the A label. Figure 2 shows an example in which two Relations R and S get partitioned using the Domain-based Partitioning.

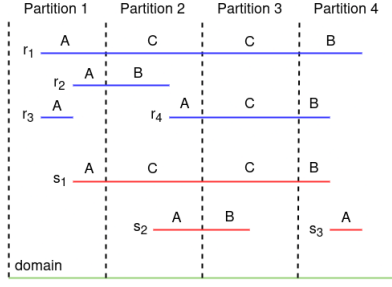


Figure 2: An example of the Domain-based Partitioning

Considering that intervals of a Relation R , for a partition i , with the category label X are represented as R_i^X , the interval join matches between relations R and S inside the partition i are computed by a set of five mini-joins: (i) $R_i^A \bowtie S_i^A$ using optFS or LEBI, (ii) $R_i^A \bowtie S_i^B$ using a group scan of S_i^B to R_i^A , (iii) $S_i^A \bowtie R_i^B$ using a group scan of R_i^B to S_i^A , (iv) $R_i^A \bowtie S_i^C$ using a Nested Loop, and (v) $S_i^A \bowtie R_i^C$ using a Nested Loop. The remaining 4 possible combinations between the labels of each partition (i.e. $\{R_i^B \bowtie S_i^C, R_i^C \bowtie S_i^B, R_i^C \bowtie S_i^C, R_i^B \bowtie S_i^B\}$) do not need to be processed, since a match produced in any of these sets will already be produced by one of the mini-joins in a previous partition containing the A counterparts of these intervals.

Furthermore, in an attempt to assign the load uniformly across all of the working threads, the partitions are created using an *Adaptive Partitioning* algorithm. The time domain is split into a number of *granules*, which are smaller in extent, but bigger in number than the final partitions and aims to combine them, in order to create partitions with approximately the same amount of A labeled intervals (as $R_i^A \bowtie S_i^A$ is usually the most expensive mini-join). The mini-joins are assigned to different threads following a *Greedy Scheduling* strategy [23]. The latest proposed Domain-based Partitioning version [9, 10] was shown to be the most efficient way to compute interval joins on multicore hardware, as it clearly outperforms the Hash-based Partitioning paradigm from prior work [40].

Figure 3 illustrates parallel interval join execution for Requests (Table 1) across all 24 cores under Domain-based Partitioning. We can clearly see an uneven load across threads. Further analysis revealed that this was due to three reasons. First, Adaptive Partitioning balances only the count of type- A intervals per partition, ignoring potential skew from expensive mini-joins involving type- B and C intervals. Second, the time-domain granules, later to be combined for the final partitions, can differ a lot in size between each other. Intervals may exist in bigger amount in specific parts of the time domain. As a result, even the number of A intervals is not identical across all partitions. Third, the Greedy Scheduling algorithm estimates mini-join cost purely from input cardinalities. However, the runtime complexity of optFS and LEBI is $|R| + |S| + |R \bowtie S|$, covering both plane-sweep scanning ($|R| + |S|$) and result reporting ($|R \bowtie S|$). When output cardinality $|R \bowtie S|$ dominates input sizes, scheduling solely on $|R| + |S|$ leads to severe workload imbalances.

2.3.4 Analysis on multisoocket CPUs. The results above show that Domain-based Partitioning, despite being tailored to create evenly-sized jobs for the available threads, fails to perform load balancing on multicore CPUs. In fact, modern servers typically pack multiple CPUs in a multisoocket configuration. In such a

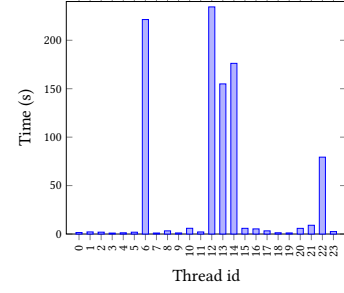


Figure 3: Exec. time of Domain-based Partitioning in optFS

setting, each CPU has its own dedicated memory controller to which DRAM modules are attached via a memory bus. The CPUs themselves are connected via a separate interconnect like Intel QuickPath, or AMD Infinity Fabric. This topology results in *Non-Uniform Memory Access* (NUMA), as memory local to a CPU can be accessed much faster compared to memory attached to another CPU that has to be accessed via the much slower interconnect. The impact of NUMA on the performance of analytical queries, and techniques to minimize this impact are well studied for relational database workloads for CPUs [32, 41] and even GPUs [34, 45]. However, no such study has been carried out for interval joins.

By default, the Domain-based Partitioning does not perform any NUMA-specific optimizations and relies on the underlying memory allocations of the operating system. The memory bandwidths for the Dig, Requests, and Emergency datasets reach 2 GB/s, 25 GB/s, and 90 GB/s, respectively, with the higher throughputs driven by the larger outputs of the Requests and Emergency workloads. These figures fall below 229 GB/s, the baseline when running the STREAM benchmark [36] on 10M-element arrays. This reduction is expected behavior because we accumulate an XOR operation rather than writing final results back to main memory, inherently generating less memory bus traffic than STREAM's write-heavy kernels. Despite this drop, the underlying stall distribution remains consistent with the patterns established in Table 2, maintaining approximately 40% remote NUMA accesses across all three datasets. As we show later (Section 4), the combination of remote accesses with high memory bandwidth utilization leads to a 1.3 $\times$ drop in performance over an equivalent NUMA-optimized algorithm under Requests. This suggests that the default NUMA allocation provided by the OS is ineffective under memory-intensive datasets.

In summary, state-of-the-art interval join algorithms fail to exploit modern CPUs at both the microarchitectural level, due to lack of vectorized instructions leading to core stalls and grouped scans causing memory stalls, and at the macro scheduling level, due to imbalances in workload scheduling and lack of NUMA-awareness.

3 Design

In this section, we present our HCIJ– Hardware-Conscious Interval Join algorithm. At its core, HCIJ uses the idea of forward scanning from optFS and introduces a new data structure named Forward Scan Map (FSMap). We first describe this data structure and the new parallel algorithm for performing forward scans based on FSMap. Later, we present various optimizations to fully exploit vectorization, improve cache utilization, and minimize NUMA impact.

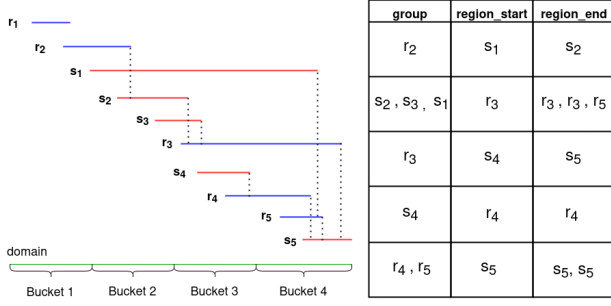


Figure 4: A Forward Scan Map for relations R (r_i tuples) and S (s_i tuples) – dotted lines denote `region_end` value

3.1 Forward Scan Map

The Forward Scan Map (FSMap) is the core data structure that HCIJ uses. An FSMap is an array, where each element consists of an object that maps a group of intervals to the memory region of future intervals that must be scanned to produce its interval join matches. Formally, we define the FSMap (M) as a set of tuples mapping an interval group from the source relation to the region of the target relation that must be scanned for it.

$$M = \{(G_i, p_{start}^{(i)}, E_i) \mid i \in \{1, \dots, k\}\} \quad (1)$$

where:

- G_i (denoted as *group* in our algorithms) is a set of consecutive records from the source relation, sorted by end endpoint, forming the i -th execution group.
- $p_{start}^{(i)}$ (denoted as *region_start*) is a pointer to the first record in the target relation where the co-scan begins for G_i .
- E_i (denoted as *region_end*) is an array of $|G_i|$ pointers to records in the target relation, aligned with G_i . For $G_i = \{g_{i,1}, \dots, g_{i,n_i}\}$, each record $g_{i,j}$ is actively scanned against the target relation from $p_{start}^{(i)}$ until $E_i[j]$ is reached.
- k is the total number of disjoint groups in M .

Figure 4 presents an example of an FSMap for two relations R and S . The relations are sorted by their start endpoint and their tuples are named according to that order. The connection between each group interval and its *region_end* is shown with dotted lines.

The first element of the FSMap maps the interval of r_2 (*group*) from relation R to the memory region spanning between intervals s_1 (*region_start*) to s_2 (*region_end*) in relation S . Note that interval r_1 will not be part of the FSMap entry due to the fact that it does not produce any result matches with future intervals from S . As described in Section 2.2, forward scans are performed from both relations, and consecutive intervals from the same relation form a group. In the second entry, group s_2, s_3, s_1 (sorted by end endpoint) will be mapped to the memory region starting from r_3 for each tuple in the group, and ending at r_3 for s_2, r_3 for s_3 , and r_5 for s_1 . This means that s_2 and s_3 will be matched only with the future interval r_3 from R and s_1 will be matched with the future intervals r_3, r_4 and r_5 . Accordingly, group r_3 will be mapped to the memory region from s_4 to s_5 and group s_4 will be mapped only with the future tuple r_4 . At last, group r_4, r_5 will be mapped to the memory region starting from s_5 and ending to s_5, s_5 for each tuple in the group, respectively. s_5 will not be mapped to any future intervals from R , as it is the last tuple in the time domain.

The benefits that we can acquire from an FSMap are several. First, obtaining the forward-scan cost in $O(1)$ time allows predicting the final result cardinality. As shown later, this enables effective, load-balanced scheduling across multiple cores that outperforms Domain-based Partitioning. Second, matches require no endpoint comparisons because each forward scan's end is predetermined. Third, the memory footprint is strictly bound by $3(|R| + |S|)$, as the worst case yields at most $|R| + |S|$ entries of three intervals each (*group*, *region_start*, *region_end*). Conversely, Domain-based Partitioning can require $(|R| + |S|) \cdot |partitions|$ memory due to interval replication across time partitions; since partition counts routinely exceed three, its memory footprint is substantially larger.

3.2 FS-based scheduling

Having described FSMap, we now present the HCIJ algorithm. Recall that the two algorithms explained in Section 2.2 propose two different versions of result reporting using a plane sweep algorithm; optFS chooses to scan the other relation, while LEBI chooses to scan an external data structure which is set up dynamically during the plane-sweep algorithm. Deriving from this, we observe an important advantage in optFS that can be exploited to parallelize it; the forward scan calls in Lines 6 and 9 of Algorithm 1 are independent of each other. This means that by knowing the starting positions of each call, we can execute them in different order or in parallel. This cannot happen with LEBI, as we need a copy of the state of the external data structure that changes constantly.

Our algorithm consists of 3 main parts; border discovery, global FSMap creation and FS-based scheduling. Considering two relations R and S , Algorithm 3 (*Border Discovery*) first executes the plane sweep algorithm (Lines 8–15), but instead of calling a forward scan in each step, it aligns and splits the intervals of the two relations into t equally sized sets (where t is the number of threads), and saves their starting and ending positions in the *borders* array (Lines 5,14,15), to pinpoint the position in which each set begins and ends. Intuitively, each set contains $(|R|+|S|)/t$ consecutive forward scan calls from Algorithm 1. As these forward scans can differ among each other in length, our goal is to schedule them carefully across available cores. To do this, we move to the second part, the creation of a global FSMap using *FSCreate* (Algorithm 4).

A global FSMap stores information about all the forward scans required to compute the interval join matches for the input relations R and S . To create it, each thread initiated at Line 3 of Algorithm 4 will read its pair of *borders* pointers and execute the plane sweep algorithm again. To save space and without loss of generality, we omit the second part of the plane sweep algorithm, where $s.start$ is the smaller endpoint after Line 19. Every step performs a group scan to discover the last future interval (i.e. the *region_end* value) of each one in the group (Lines 8–17). We first gather and sort the group of consecutive intervals from the same relation by end endpoint to perform a group scan (Line 9). Then, we read each group interval in a loop (Lines 10–17). To reach the last future interval faster, we make use of the Bucket Index data structure. The time domain is split into a number of equally sized buckets (as also shown in Figure 4) and we scan directly the Bucket B in which the end endpoint of the group interval is located, until we reach the last future interval that overlaps with it (Lines 11–14). For every group interval in the group scan, the entry of the global FSMap (m) is updated with the current

Algorithm 3: Border Discovery

Input: Relation R , Relation S , number of threads t
Output: all intersecting interval pairs $(r, s) \in R \times S$

```

1 sort  $R$  and  $S$  by start endpoint using  $t$  threads ;
2  $r \leftarrow$  first interval in  $R$  ;
3  $s \leftarrow$  first interval in  $S$  ;
4 create  $borders$  array of size  $t+1$ ;
5  $borders[0] \leftarrow \{r, s\}$  ;
6  $cnt \leftarrow 0$  ;
7 while  $r \neq null$  or  $s \neq null$  do
8   if  $s == null$  or  $(r \neq null \text{ and } r.start < s.start)$  then
9      $r \leftarrow$  next interval in  $R$  ;
10  else
11     $s \leftarrow$  next interval in  $S$  ;
12     $cnt \leftarrow cnt + 1$  ;
13    if  $cnt \% ((|R| + |S|) / t) == 0$  then
14       $borders[cnt / ((|R| + |S|) / t)] \leftarrow \{r, s\}$  ;
15  $borders[t] \leftarrow \{null, null\}$  ;
16  $M_{global} = FSCreate(R, S, borders, t)$  ;
17  $J = FS\_based\_scheduling(M_{global}, t)$  ;
18 return  $J$  ;
```

group interval and its forward scan extent (Lines 15–17). In case the entry m contains at least one valid *group* value at the end of the current group scan, the FMap will receive it as a new entry (Lines 22–23).

To describe FSCreate with an example, consider again Figure 4 and the case of the first two intervals r_1 and r_2 . These two consecutive intervals will form a group scan and since they belong in relation R , they will follow the Lines 7–18 in the plane sweep loop of Algorithm 4. As s_1 is the first interval from Relation S that follows the group in the time domain, it will be added in the *region_start* field of m (Line 8). Later, the group will be sorted by end endpoint (Line 9). In the forward scan between Lines 11 and 14, interval r_1 , whose end endpoint is in Bucket 1 will read s_1 and will stop immediately, as $r_1.end < s_1.start$. Entry m will not be updated. The for loop will read the next tuple r_2 . As $r_2.end$ exists in Bucket 2, the forward scan will start from s_2 and end at s_3 . Entry m will receive r_2 as *group* and $(s_3 - 1 = s_2)$ as *region_end* at Lines 16–17. The rest of the FMap entries will be filled according to the same logic.

After creating the global FMap for relations R and S , we have full information of the total interval join cost and can start the scheduling of the forward scan calls by calling Algorithm 5 (*FS-based scheduling*). The algorithm splits M_{global} into t smaller ones (equal to the number of threads that we use). Considering that a load of an FMap entry m is computed by $\sum_{i=1}^{|m.group|} m_i.region_end - m_i.region_start + 1$ (meaning the total size of the memory region to be scanned), our goal is to create t FMaps of nearly equal load. To do this, we use a very simple scheduling strategy that reads M_{global} entry-by-entry to assign each one to the FMap with the smallest load (Lines 1–6). Because a new entry is always assigned to the thread currently possessing the minimum total load, the difference in load between the most loaded thread and the least loaded thread can never exceed the cost of the single largest FMap entry. Given that the number of tuples in an entry is strictly capped by a maximum group size (an optimization we explain later in Section 3.3.2), the workload

Algorithm 4: FSCreate

Input: Relation R , Relation S , Pointers array $borders$, number of threads t
Output: FMap M_{global} of relations R and S

```

1 create bucket indexes  $BI^R, BI^S$  ;
2 create empty map  $M_{global}$  ;
3 foreach  $j$  in  $0, 1, \dots, t-1$  do // in parallel
4    $\{r, s\} \leftarrow borders[j]$  ;
5   while  $r \neq borders[j+1].r$  or  $s \neq borders[j+1].s$  do
6     create FMap entry  $m$  ;
7     if  $s == borders[j+1].s$  or  $(r \neq borders[j+1].r$ 
8       and  $r.start < s.start)$  then
9        $m.region\_start \leftarrow s$  ;
10      create and sort group  $G^R$  ;
11      foreach  $r'$  in  $G^R$  do
12         $B \leftarrow$  bucket of  $r'.end$  in  $BI^S$  ;
13         $s' \leftarrow$  first interval in  $B$  ;
14        while  $s' \neq null$  and  $s'.start < r'.end$  do
15           $s' \leftarrow$  next interval in  $S$  ;
16          if  $s' - 1 \geq s$  then
17             $m.group.add(r')$  ;
18             $m.region\_end.add(s' - 1)$  ;
19       $r \leftarrow$  next interval in  $R$  ;
20   else
21     ... ; // Lines 8-17 from  $s$  to  $r$ 
22      $s \leftarrow$  next interval in  $S$  ;
23   if  $m.group \neq null$  then
24      $M_{global}.insert(m)$  ;
25 wait until all threads finish; // synchronization
26 return  $M_{global}$  ;
```

across threads remains quite steady relative to the total interval join workload or relation size, ensuring robust load balancing in practice.

Considering the example of Figure 4, Figure 5 presents the FMaps produced after FS-based scheduling, using 2 working threads. As the loads are always computed on-the-fly, we show them with red color next to each entry and FMap. The forward scan of r_2 with a load of two will be first inserted in the empty M_1 . Then, the forward scan of s_2, s_3, s_1 with a total load of five will be inserted in the empty M_2 . As the next two entries have a load of two and one, they will all be inserted in M_1 , evening its load to the one of M_2 . The last entry of group r_4, r_5 will also be inserted to M_1 randomly, as the current M_1 load is equal to M_2 , leading to a total load of seven. As explained in previous section, a global FMap requires $(|R| + |S|) \cdot 3$ memory space to store its information. Considering the copies created to split the FMap into t smaller ones, the total memory complexity of the algorithm can rise up to $(|R| + |S|) \cdot 6$. Finally, each thread simply reads its FMap, scans the memory location of its entries, and reports the result tuples. Since the end of every forward scan is known, each FMap entry is treated as a group scan in Lines 10–15 and its matches are reported without any endpoint comparisons.

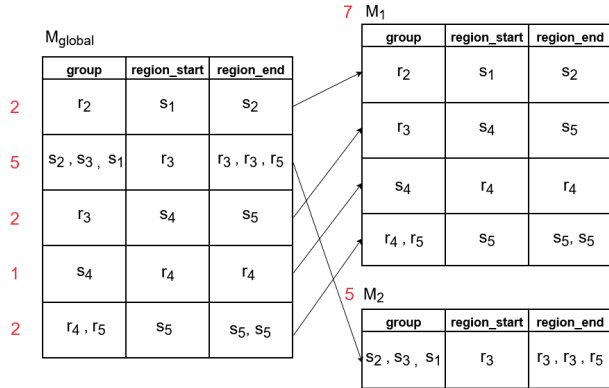
Self-tuning mechanism in HCIJ When forward scans are short, worker threads can skip *FSCreate* and directly evaluate join

Algorithm 5: FS-based scheduling**Input:** FSMap M_{global} , number of threads t **Output:** all intersecting interval pairs $(r, s) \in R \times S$

```

1 create  $t$  empty maps  $M_0, M_1, \dots, M_{t-1}$ ;
2 foreach  $m$  in  $M_{global}$  do
3    $current\_cost \leftarrow \sum_{i=1}^{|m.group|}$ 
4    $m_i.region\_end \leftarrow m_i.region\_start + 1$ ;
5    $M_i \leftarrow min\_cost(M_0, M_1, \dots, M_{t-1})$ ;
6    $M_i.insert(m)$ ;
7    $M_i.cost += current\_cost$ ;
7  $J \leftarrow \emptyset$ ;
8 foreach  $j$  in  $0, 1, \dots, t-1$  do           // in parallel
9   foreach entry  $m$  in  $M_j$  do
10     $b' \leftarrow m.region\_start$ ;
11    foreach  $\{a_i \in m.group, e_i \in m.region\_end\}$  do
12      while  $b' \leq e_i$  do
13        foreach  $a_j \in m.group, j \geq i$  do
14           $J \leftarrow J \cup \{(a_j, b')\}$ ;
15           $b' \leftarrow next\ interval$ ;
16 wait until all threads finish;           // synchronization
17 return  $J$ ;

```

**Figure 5:** Splitting M_{global} into 2 FSMaps.

matches using the *borders* array from Algorithm 3. To determine whether FS-based scheduling is worthwhile, we adapt the selectivity heuristic from optFS [10], which originally decides whether to apply grouping and bucket indexing for short-lived intervals.

3.3 Hardware-conscious optimizations

As we show later, HCIJ solves the load-imbalance issues of the Domain-based Partitioning. We now describe various optimizations to address other issues described in Section 2.

3.3.1 Vectorized Forward Scans. Recall that LEBI and even more, optFS could not exploit the vectorization capability of modern CPUs as it was not explicitly designed for SIMD parallelism, and compilers could not perform automatic vectorization due to data dependencies. In HCIJ, we overcome this problem by explicitly redesigning and reimplementing the forward scan to exploit SIMD parallelism. SIMD instructions are particularly useful in the case of linear memory accesses [25]. Hence, to maximize the

performance achievable with SIMD load instructions, we store the data in a columnar layout [26]. This way, we are able to load the endpoints of consecutive future intervals to SIMD registers faster.

Our strategy works as follows. In the beginning of a forward scan, we copy each group interval into a SIMD register k times, where k is the amount of intervals that can fit in a SIMD register. For example, a 256-bit SIMD register would require the copy of a 64-bit interval four times. This SIMD register will retain its values for the rest of the forward scan, just like variables in a scalar execution. While future intervals are scanned, interval matches are reported. Instead of reporting matches one-by-one between the group and the future intervals, we load every k future intervals in another SIMD register, and perform interval matching in a vectorized fashion. The efficiency of our strategy is based on the fact that the group interval needs to be loaded in a SIMD register only once, at the beginning of the forward scan, while the loading of the future intervals takes advantage of the linear access of our columnar layout.

Our version of vectorized forward scans can be applied on any instruction set and on different modes. For example, our forward scan can be used to copy the results in a data structure, or it can compute the real overlap between the two matches directly. In practice, vectorized forward scans require a set of four future intervals to use them in a SIMD register. Thus, our real implementation integrates explicit vectorization with loop unrolling (also used in optFS and LEBI) to improve the effectiveness of SIMD vectorization.

3.3.2 Cache-sized Grouping. Recall that the second major source of stalls in optFS was memory stalls. Also, recall that the grouping optimization proposed in [9, 10] was applied with a goal to reduce the number of scans performed on future intervals. In Section 2, we showed that the memory stalls (and specifically, those of the Emergency dataset) were due to large-sized group scans. To avoid such issues, we apply an upper bound for the maximum possible group size, in a similar way to the already existing one in LEBI. This provides us two benefits. First, it provides a balance between memory stalls on future interval scans caused by not grouping, and memory stalls on group intervals from aggressive grouping. Second, as our parallel algorithm schedules multiple group scans directly to the CPU cores, unusually large group scans can create load management issues to our scheduling algorithm. Splitting big groups into smaller sets can provide a more precise scheduling.

3.3.3 NUMA-awareness. Finally, we address NUMA-awareness on multsocket CPUs. Because state-of-the-art techniques lack NUMA-specific allocation strategies (Section 2), memory requests frequently target remote memory. Our goal is to pin threads and allocate data structures to matching CPU-DRAM nodes so that every thread accesses strictly local memory.

To achieve NUMA locality, many database systems partition data structures across sockets [32], restructure the algorithm, and pin threads to limit cross-socket traffic. In HCIJ, partitioning the time domain across sockets would reproduce the work-distribution skew of Domain-based Partitioning (Section 2.3). Instead, leveraging the multi-terabyte memory capacities typical of modern servers, we replicate both input relations across all CPU sockets. Because HCIJ avoids data partitioning, NUMA-awareness simply requires setting FSMap pointers to the local CPU socket's replica. While this increases memory consumption for FS-based scheduling (Section 3.2), the footprint grows only

from $(|R| + |S|) \cdot 6$ to $(|R| + |S|) \cdot 8$ on dual-socket or $(|R| + |S|) \cdot 10$ on quad-socket systems. With modern servers typically hosting far more than 10 threads, this remains dramatically lower than Domain-based Partitioning's $(|R| + |S|) \cdot |threads|$ footprint.

Moreover, as shown in Table 2 and Section 4, NUMA optimizations yield negligible gains—yet incur replication overhead—on datasets with few memory stalls. To prevent unnecessary overhead, HCIJ runs a lightweight, one-time offline probe to measure the dataset's average group size, replicating relations only when this value falls below a threshold (empirically determined to be ten on our servers). Since grouping is deployed for costly joins dominated by extended forward scans, this profiling cost is negligible.

4 Evaluation

In this section, we evaluate HCIJ. We use the same server from Section 2.3.2 (hyper-threading disabled). All methods were implemented in C++ (g++ v11.4: -O3, -lnuma, -ltbb, -mavx512f). Following [9, 10, 40], intervals use 64-bit endpoints, data reside in main memory, and workloads compute an *XOR* sum of start attributes. To further investigate the performance of our design on different architectures, we also conducted the same analysis on a 16-core ARM Graviton3 CPU, a 192-core Intel Xeon 7640E CPU, and a 64-core quad-socket RISC-V CPU. Specific details about each are given inline later in this section with relevant results.

Real-world Datasets We use the three real world datasets described in Table 1. In addition, we also report the results under the Taxis dataset, which is the largest real-world dataset used in previous work [10] and contains shorter intervals. Relations are created after cleaning the original dataset from invalid records and then choosing each tuple at a time randomly for each generated relation.

Competitors We compare HCIJ against state-of-the-art methods optFS [10], LEBI [40] (codebase kindly given from the creators) and two extra variations of them; bgFS [9] (an older version of optFS), and OMJ [19] (a double-plane sweep version of optFS).

Software Setup Aligning with prior work [9, 10], we use 1k bucket indexes, loop unrolling factor of 32 and identical Domain-based Partitioning granules. The optFS tuning threshold is 100. HCIJ dynamically selects between FS-based scheduling or direct execution using the *borders* array (Algorithm 3) with a higher threshold, as both paths support vectorization (in contrast to optFS). `tbb::parallel_sort` is used for parallel sorting.

4.1 HCIJ and state-of-the-art interval joins

Figure 6 compares the end-to-end runtime of all five approaches across various input-size ratios, using all 24 physical cores. Run-times include sorting, scheduling, and join execution. We executed each experiment five times and reported the mean value across these runs to mitigate transient fluctuations. HCIJ consistently outperforms all baselines across all datasets, reducing execution time by 1.2× to 3.2× over optFS and LEBI. It achieves an average 1.8× speedup on Dig and Emergency and over 2× on Requests—where the competing parallel strategy fails to adapt to the dataset (Figure 3). Notably, at $|R| = |S|$, HCIJ completes in 67 seconds, 3.2× faster than optFS (214 seconds). On Taxis, the speedup is more modest (1.2×) due to small selectivity (0.00005) and most of the time was spent by parallel sorting.

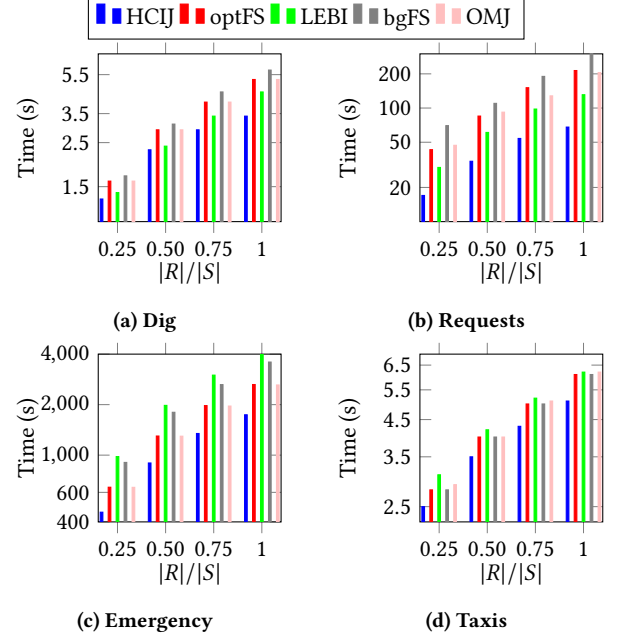


Figure 6: HCIJ against competitors – End to end mean execution time (s) – 24 threads

Comparison with indexes As we mentioned in Section 2.1, index-based methods are not strong competitors for large, unindexed interval joins. This is the reason why prior work does not compare them to non-indexed ones [7, 10, 13, 16]. For the sake of completeness, we also note that when comparing HCIJ against HINT [16], the state-of-the-art index for overlap queries, it performs 2-10× faster for all datasets. As interval indexes are usually used to index and perform multiple and cheaper operations like searches and updates on ranges, an interesting direction for future work could be the re-design of the FSMap structure as a tree-index for one or two relations to support these operations.

In summary, various HCIJ optimizations described earlier enable it to consistently outperform state-of-the-art solutions under all datasets. In addition to Figure 6, we also present Table 3, which for the case of $|R|/|S| = 1$, demonstrates the increase we get in performance by applying each of the optimizations to our three datasets from Table 1. The results before the use of explicit vectorization derive from an auto-vectorized version used from our codebase that resembles a lot the one from [9, 10, 40]. We see the benefits that we receive from FS-based scheduling and explicit vectorization (outperforming competitors from Figure 6 without caching or NUMA optimizations for Dig and Requests). Cache grouping solves memory stalls problems from vector register pressure at Emergency, while NUMA replicas boost further the achieved performance at Requests. To better understand how each optimization in HCIJ contributes to this overall improvement, the following sections analyze each one in isolation, referring back to Table 3 when necessary.

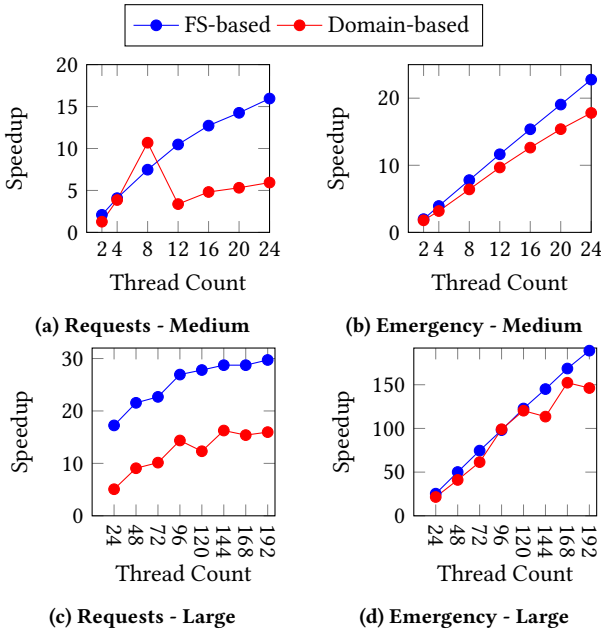
4.2 FS-based scheduling

We first investigate the performance of FS-based scheduling and present our results from comparing it to the Domain-based Partitioning. To provide a fair comparison, we implement Domain-based Partitioning in our codebase and apply all hardware-based optimizations described in Section 3.3 to it (effectively comparing

Table 3: Improvement acquired on HCIJ per-mechanism – $|R|=|S|$ – 24 threads

Dataset	Dig	Requests	Emergency
FS-based scheduling	4.6 s	97 s	2581 s
+ explicit SIMD	3.4 s	86 s	6490 s
+ cache-sized grouping	3.4 s	86 s	1757 s
+ NUMA replicas	3.4 s	67 s	1757 s

both versions after applying all optimizations, like the last line of Table 3). Both speedup lines are produced considering the same single-threaded execution. We compare their scalability in terms of the speedup they can achieve. In addition to the 24-core server (referred as *Medium*), we also report results from a server equipped with a 192-core Intel Xeon 7640E CPU and 256GB of memory (referred as *Large*). Figure 7 demonstrates the speedup of both approaches for the real-world datasets of Requests and Emergency.

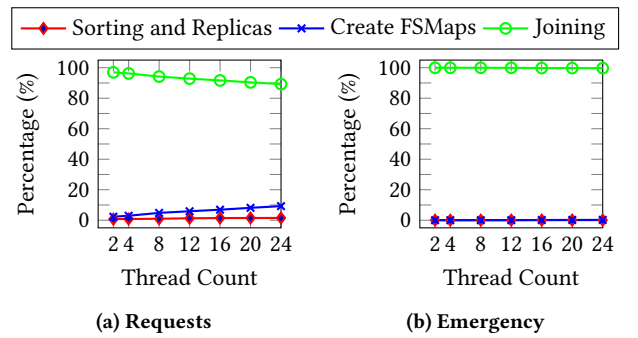
**Figure 7: FS-based scheduling vs Domain-based Partitioning – Speedup Comparison – $|R|=|S|$**

We first analyze the results for our Medium server. In both datasets, HCIJ’s FS-based scheduling algorithm scales better than the Domain-based Partitioning. Specifically for Requests, Domain-based Partitioning stops scaling after eight threads. The root cause of this issue is the Adaptive Partitioning strategy. As described in Section 2, the Adaptive Partitioning algorithm splits the time domain in a number of granules containing intervals from the *A* group with the goal of creating partitions of equally-sized *A* interval groups. However, in the case of Requests, the granules created have extreme differences in terms of the intervals that exist in each of them, as there are parts in the time domain with many more intervals compared to other parts. As a result, threads get mini-joins of extremely different loads. In addition,

the number of threads determines different partition borders in the time domain for every execution, making the partitioning unstable; as can be seen, eight threads produce balanced partitions, which together with the reduced number of comparisons from nested loop mini-joins give a speedup higher than eight. In contrast, 12, 16, 20 and 24 threads result in unbalanced partitions. As explained in Section 3.2, FS-based scheduling distributes forward scans across cores depending on the amount of results that each forward scan has to report. Hence, the scheduling logic is independent of the interval distribution in the time domain. As a result, it scales very well under both datasets.

A similar superior performance is also observed on Large. Despite having more partitions due to bigger thread count, the Domain-based partitioning still struggles to reach the efficiency of FS-based Scheduling. For both datasets, we see that scaling sometimes decreases, because the partition borders change depending on the current thread count and the thread jobs are not as balanced as with FS-based scheduling. We also observe that the speedup achieved for Emergency is much bigger (and nearly equal to the thread count) than the one in Requests. As both strategies scan their relations or FMap entries per-group, small group scans across multiple cores cause additional memory stalls while processors fetch data concurrently. Regardless of this observation, FS-based scheduling remains superior in nearly all cases.

To provide more information about the efficiency of our algorithm, we also present an execution breakdown of its phases in Figure 8. We present the breakdown only for our *Medium* machine due to space restrictions. We briefly note that the same behavior was also observed for *Large*. As we can see, the joining phase always dominates the execution time. Time spent on parallel sorting is always negligible. The time spent on FS-based scheduling is considerably bigger for Requests. This is due to the smaller group sizes (i.e. more memory to be scanned to reach the end of a forward scan at Algorithm 4). However, since we never report the results but only scan their endpoints, the time to create the FMaps for each thread remains much smaller than the time required to report the results.

**Figure 8: FS-based scheduling – Execution breakdown**

The reason why FS-based Scheduling can create more precise jobs than the Domain-based Partitioning is that it is not dependent on the interval distribution in the time domain. To further demonstrate this difference, we also test the two approaches using synthetic data. Each relation we create consists of 10M tuples, with start endpoints in the fixed range of $[0, 10^9]$. Each interval receives a random length between 1M to 100M. Figure 9 demonstrates the results of our experiment for high skew. In

each execution, we generate start endpoints where each one has a probability P to appear at a specific spot in the time domain. In case the start endpoint does not get assigned to this spot, its position in the time domain is decided randomly. As the spot becomes even more populated (high skew), the Domain-based Partitioning strategy fails to create partitions of equal size, as all intervals exist in the same granule. In contrast, FS-based scheduling ignores the interval distribution in the time domain and carefully schedules forward scans across all cores. The difference in computation time is even bigger than the one presented in Requests, surpassing 10 $\times$.

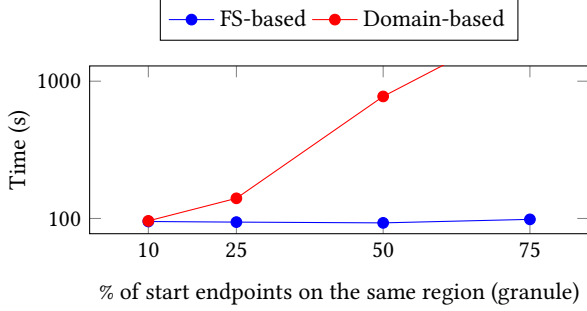


Figure 9: FS-based scheduling vs Domain-based Partitioning with high skew

4.3 Vectorization

We next investigate the effectiveness of vectorization in forward scans. Table 4 demonstrates the differences in computation time between the state-of-the-art, auto-vectorized version of optFS and our vectorized version for various datasets. To isolate the impact of vectorization, we present single-threaded execution times using 256-bit and 512-bit registers for our strategy (Section 3.3.1), using the x86_64 ISA of our Intel machine, as well as the auto-vectorization performed by the compiler from the optFS codebase. As we show in Table 3 and also in next Section, setting an upper bound to the maximum group size is pivotal for vectorized performance. We thus keep this mechanism active in all of our experiments.

As seen in Table 4, HCIJ’s vectorized forward scan is always superior in terms of performance compared to the auto-vectorized version of optFS, with the 256-bit SIMD implementation providing up to 1.3 $\times$ speedup, and the 512-bit SIMD implementation providing up to 1.63 $\times$ speedup, over the auto-vectorized code. This demonstrates the benefit that can be achieved by redesigning interval joins to exploit SIMD vectorization explicitly. To evaluate HCIJ’s portability to other architectures, we extended it with support for NEON intrinsics. We used a 16-core ARM Graviton3 instance to evaluate HCIJ and optFS. The second half of Table 4 shows that HCIJ outperforms optFS by a factor of 1.25 $\times$ on all datasets. The observed improvement is smaller than the x86_64 case as Graviton3 NEON uses smaller 128-bit sized registers. However, this result shows the portability of our solution to different architectures.

4.4 Cache-sized grouping

We next evaluate cache-friendly grouping efficiency (Section 3.3.2) via a sensitivity analysis on the Emergency dataset, whose interval join produces groups of up to 70K tuples (Section 2.3.2). Considering that our test machine’s 48 KiB L1d cache

Table 4: Vectorized Forward Scans – 1 thread (s)

Intel Xeon Gold 5317	Dig	Requests	Emergency
Auto-vectorization [9, 10, 40]	93	1413	55K
256-bit SIMD vectorization	73	1085	44K
512-bit SIMD vectorization	57	1069	40K

ARM Graviton3	Dig	Requests	Emergency
Auto-vectorization [9, 10, 40]	228.5	781.6	150K
Vectorization with NEON	170.5	652.5	122.4K

can hold up to 6k 64-bit integers, we vary the forward-scan group size from 10 to 70K to test both sub-cache and cache-exceeding sizes.

Figure 10 reports execution times and LLC misses per tuple using vectorization and 24 threads. Increasing group size from 10 to 100 yields a 3 $\times$ improvement and performance plateaus beyond 100 as HCIJ saturates, reaching its peak performance. However, when we surpass the group size of 10k, the optimization results in a rapid performance drop. By 70k, grouping performs identically to no grouping; groups of stable intervals no longer fit in cache, causing DRAM stalls reflected by the surge in LLC misses (Figure 10b), which mirrors runtime (Figure 10a).

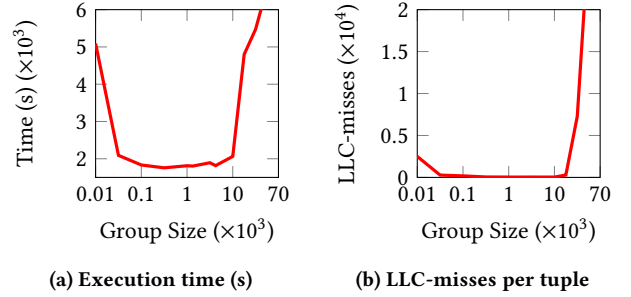


Figure 10: Group size impact for Emergency – 24 threads

These findings highlight the need to tailor grouping to hardware. Unlike prior work [40] that sized groups to the L1d capacity, peak performance here occurs well before the 6K limit. This discrepancy arises because the cache is shared with the future relation’s tuples, while limited SIMD registers expose memory bottlenecks sooner. This also accounts for the performance degradation under unbound grouping in Table 3. Furthermore, smaller groups provide better load balancing across threads in FS-based scheduling. We therefore set the group size to the performance plateau’s lower knee (500 based on this benchmark).

4.5 NUMA-awareness and Memory Footprint

In this section, we evaluate our NUMA-aware design and its memory footprint. Because NUMA impact correlates with memory stalls, we test two datasets (Figure 11): Requests (high stalls, small groups) and Dig (low stalls, large groups). We compare a “NUMA-aware” setup (thread-pinning with relation replicas) against an “OS” baseline (thread migration without replicas). For Requests, once execution crosses the 12-core socket boundary, remote memory fetches cause the “OS” baseline to lag, allowing the NUMA-aware approach to achieve a 1.3 $\times$ speedup at 24 threads. Conversely, Dig sees no difference since large groups naturally mask memory pipeline stalls. Thus, selectively generating NUMA

replicas only for datasets with small groups efficiently boosts performance.

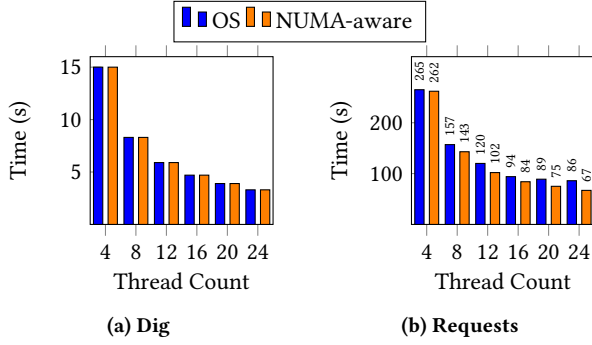


Figure 11: NUMA impact on HCIJ (s)

We next provide a memory footprint analysis with experiments where NUMA replicas are existent or non-existent. We measured memory consumption using our *Medium* and *Large* machines from Section 4.2 for the case of two NUMA nodes and also using a RISC-V-based, Milk-V Pioneer machine with four NUMA-nodes. As we want to provide a fair comparison, we refrain from using Requests, as the Domain-based Partitioning strategy is not effective with it (Figure 3). We therefore present our memory analysis for Dig and Emergency. Figure 12 shows the maximum memory resident throughout the join for each dataset, while increasing the number of threads (number of partitions for the Domain-based Partitioning). We observe that for the case of the Dig dataset, which contains intervals that are shorter and do not result in many replications, the Domain-based Partitioning manages to compete well against the FS-based scheduling. While the NUMA-unaware version is generally less memory intensive, adding more replicas results in a slightly higher memory usage. On the other hand, Emergency results reflect the worst-case scenario, as most of the intervals are replicated in all partitions. Replicas that are spread across the whole time domain result in memory demands of more than 80GB to compute a join of relations that are less than 3GB in size. In contrast, FS-based scheduling requires only 8.8GB of space for the case of the 4-NUMA replicas and even less for the other two cases.

We ultimately conclude that while there could be cases in which the NUMA replicas could result in a bigger footprint, FS-based scheduling is still a safer choice because a) it avoids pathological scenarios of extreme interval replication, b) the relation replication will happen in scenarios of low selectivity with small group sizes and can be easily disregarded without hurting performance in case of limited available memory, c) the memory footprint for selective joins is zero after Algorithm 3, and most importantly, d) it provides a faster join computation.

5 Conclusion and Future Work

Current approaches to interval join processing often overlook the potential performance gains achievable by fully exploiting the underlying hardware. In this paper, we introduced a novel parallel algorithm that significantly boosts interval join performance while maintaining a low memory footprint. Our design takes advantage of modern CPUs and the structure of their memory hierarchy. Experimental results demonstrate that our hardware-aware approach outperforms existing interval join methods. Looking forward, we aim to further extend our design to alternative ways

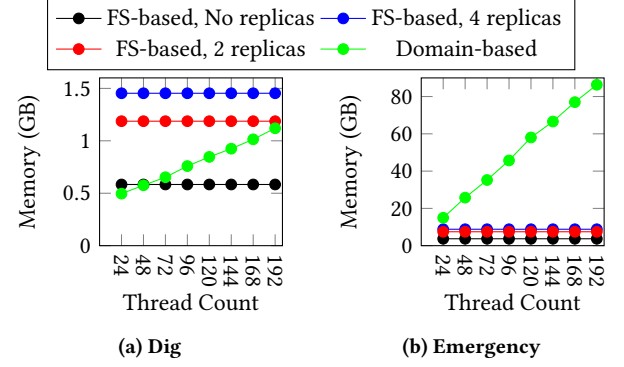


Figure 12: FS-based scheduling vs Domain-based Partitioning – Memory Footprint Comparison

of creating and vertically scanning Forward Scan Maps on more interval operations such as Allen relations, and investigating support for interval joins over dynamic data with insertions and updates.

Artifacts

The source code of HCIJ is available at <https://gitlab.eurecom.fr/dislalab/hcij>.

Acknowledgments

This work was partially supported by the French government through the PEPR MolecuArXiv (France 2030 program), and European Union’s Horizon Europe research and innovation programme under grant agreement No 101136962 (NextGen).

References

- [1] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. 1999. DBMSs on a Modern Processor: Where Does Time Go?. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 266–277.
- [2] James F. Allen. 1983. Maintaining knowledge about temporal intervals. *Commun. ACM* 26, 11 (Nov. 1983), 832–843. doi:10.1145/182.358434
- [3] Lars Arge, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel, and Jeffrey Scott Vitter. 1998. Scalable Sweeping-Based Spatial Join. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 570–581.
- [4] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. 1996. An asymptotically optimal multiversion B-tree. *The VLDB Journal* 5, 4 (Dec. 1996), 264–275. doi:10.1007/s007780050028
- [5] Lawrence Benson, Richard Ebeling, and Tilmann Rabl. 2023. Evaluating SIMD Compiler-Intrinsics for Database Systems. In *Proceedings of the International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*.
- [6] Maximilian Böther, Lawrence Benson, Ana Klimovic, and Tilmann Rabl. 2023. Analyzing Vectorized Hash Tables across CPU Architectures. *Proc. VLDB Endow.* 16, 11 (July 2023), 2755–2768. doi:10.14778/3611479.3611485
- [7] Panagiotis Bours, George Christodoulou, Christian Rauch, Artur Titkov, and Nikos Mamoulis. 2025. Querying Interval Data on Steroids. *IEEE Transactions on Knowledge and Data Engineering* 37, 10 (2025), 6120–6134. doi:10.1109/TKDE.2025.3597399
- [8] Panagiotis Bours, Konstantinos Lampropoulos, Dimitrios Tsitsigkos, Nikos Mamoulis, and Manolis Terrovitis. 2020. Band Joins for Interval Data. In *EDBT*. 443–446.
- [9] Panagiotis Bours and Nikos Mamoulis. 2017. A forward scan based plane sweep algorithm for parallel interval joins. *Proc. VLDB Endow.* 10, 11 (Aug. 2017), 1346–1357. doi:10.14778/3137628.3137644
- [10] Panagiotis Bours, Nikos Mamoulis, Dimitrios Tsitsigkos, and Manolis Terrovitis. 2021. In-Memory Interval Joins. *The VLDB Journal* 30, 4 (April 2021), 667–691. doi:10.1007/s00778-020-00639-0
- [11] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. 1993. Efficient processing of spatial joins using R-trees. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (Washington, D.C., USA) (SIGMOD '93)*. Association for Computing Machinery, New York, NY, USA, 237–246. doi:10.1145/170035.170075

- [12] Francesco Cafagna and Michael H. Böhlen. 2017. Disjoint interval partitioning. *The VLDB Journal* 26, 3 (June 2017), 447–466. doi:10.1007/s00778-017-0456-7
- [13] Matteo Ceccarello, Anton Dignös, Johann Gamper, and Christina Khnaisser. 2023. Indexing Temporal Relations for Range-Duration Queries. In *Proceedings of the 35th International Conference on Scientific and Statistical Database Management* (Los Angeles, CA, USA) (SSDBM '23). Association for Computing Machinery, New York, NY, USA, Article 3, 12 pages. doi:10.1145/3603719.3603732
- [14] Reynold Cheng, Sarvjeet Singh, Sunil Prabhakar, Rahul Shah, Jeffrey Scott Vitter, and Yuni Xia. 2006. Efficient join processing over uncertain data. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management* (Arlington, Virginia, USA) (CIKM '06). Association for Computing Machinery, New York, NY, USA, 738–747. doi:10.1145/1183614.1183719
- [15] Georgios Christodoulou. 2023. *Interval Data Management in Main Memory*. Ph.D. Dissertation. University of Ioannina.
- [16] George Christodoulou, Panagiotis Boursos, and Nikos Mamoulis. 2022. HINT: A Hierarchical Index for Intervals in Main Memory. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) (SIGMOD '22). Association for Computing Machinery, New York, NY, USA, 1257–1270. doi:10.1145/3514221.3517873
- [17] George Christodoulou, Panagiotis Boursos, and Nikos Mamoulis. 2024. LIT: Lightning-fast In-memory Temporal Indexing. *Proc. ACM Manag. Data* 2, 1, Article 20 (March 2024), 27 pages. doi:10.1145/3639275
- [18] Anton Dignös, Michael H. Böhlen, and Johann Gamper. 2014. Overlap interval partition join. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) (SIGMOD '14). Association for Computing Machinery, New York, NY, USA, 1459–1470. doi:10.1145/2588555.2612175
- [19] Anton Dignös, Michael H. Böhlen, Johann Gamper, Christian S. Jensen, and Peter Moser. 2021. Leveraging range joins for the computation of overlap joins. *The VLDB Journal* 31, 1 (Aug. 2021), 75–99. doi:10.1007/s00778-021-00692-3
- [20] Jost Enderle, Matthias Hampel, and Thomas Seidl. 2004. Joining interval data in relational databases. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France) (SIGMOD '04). Association for Computing Machinery, New York, NY, USA, 683–694. doi:10.1145/1007568.1007645
- [21] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA database - An architecture overview. *IEEE Data Eng. Bull.* 35 (03 2012), 28–33.
- [22] D. Gao, Christian Søndergaard Jensen, R.T. Snodgrass, and M.D. Soo. 2005. Join Operations in Temporal Databases. *VLDB Journal* 14, 1 (2005), 2–29.
- [23] R. L. Graham. 1969. Bounds on Multiprocessing Timing Anomalies. *SIAM J. Appl. Math.* 17, 2 (March 1969), 416–429. doi:10.1137/0117039
- [24] Himawan Gunadhi and Arie Segev. 1991. Query Processing Algorithms for Temporal Intersection Joins. In *Proceedings of the Seventh International Conference on Data Engineering*. IEEE Computer Society, USA, 336–344.
- [25] Dirk Habich and Johannes Pietrzyk. 2024. SIMDified Data Processing - Foundations, Abstraction, and Advanced Techniques. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) (SIGMOD '24). Association for Computing Machinery, New York, NY, USA, 613–621. doi:10.1145/3626246.3654694
- [26] Max Heimeel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-oblivious parallelism for in-memory column-stores. *Proc. VLDB Endow.* 6, 9 (July 2013), 709–720. doi:10.14778/2536360.2536370
- [27] Xiao Hu, Stavros Sintos, Junyang Gao, Pankaj K. Agarwal, and Jun Yang. 2022. Computing Complex Temporal Join Queries Efficiently. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) (SIGMOD '22). Association for Computing Machinery, New York, NY, USA, 2076–2090. doi:10.1145/3514221.3517893
- [28] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter Michael Fischer, Donald Kossmann, Franz Färber, and Norman May. 2013. Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) (SIGMOD '13). Association for Computing Machinery, New York, NY, USA, 1173–1184. doi:10.1145/2463676.2465293
- [29] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.* 11, 13 (Sept. 2018), 2209–2222. doi:10.14778/3275366.3284966
- [30] Krishna Kulkarni and Jan-Eike Michels. 2012. Temporal features in SQL:2011. *SIGMOD Rec.* 41, 3 (Oct. 2012), 34–43. doi:10.1145/2380776.2380786
- [31] T. Y. Cliff Leung and Richard R. Muntz. 1992. Temporal Query Processing and Optimization in Multiprocessor Database Machines. In *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings*, Li-Yan Yuan (Ed.). Morgan Kaufmann, 383–394. <http://www.vldb.org/conf/1992/P383.PDF>
- [32] Yinan Li, Ippokratis Pandis, René Müller, Vijayshankar Raman, and Guy M. Lohman. 2013. NUMA-aware algorithms: the case of data shuffling. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. <https://www.microsoft.com/en-us/research/publication/numa-aware-algorithms-the-case-of-data-shuffling/>
- [33] Hongjun Lu, Beng Chin Ooi, and Kian-Lee Tan. 1994. On Spatially Partitioned Temporal Join. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 546–557.
- [34] Tobias Maltenberger, Ivan Ilic, Ilin Tolovski, and Tilmann Rabl. 2022. Evaluating Multi-GPU Sorting with Modern Interconnects. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) (SIGMOD '22). Association for Computing Machinery, New York, NY, USA, 1795–1809. doi:10.1145/3514221.3517842
- [35] Nikos Mamoulis. 2011. *Spatial Data Management*. Vol. 3. doi:10.2200/S00394ED1V01Y201111DTM021
- [36] John McCalpin. 1995. Memory bandwidth and machine balance in high performance computers. *IEEE Technical Committee on Computer Architecture Newsletter* (12 1995), 19–25.
- [37] Gabriele Mencagli, Yuriy Rymarchuk, and Dalvan Griebler. 2025. PPOIJ: Shared-Nothing Parallel Patterns for Efficient Online Interval Joins over Data Streams. In *DEBS '25: Proceedings of the 19th ACM International Conference on Distributed and Event-based System*. 51–61. doi:10.1145/3701717.373054
- [38] Alexandru Nicolau. 1985. Loop quantization: Unwinding for fine-grain parallelism exploitation. In *Dept. of Computer Science, Cornell University*.
- [39] Danila Piatov, Sven Helmer, Anton Dignös, and Fabio Persia. 2021. Cache-efficient sweeping-based interval joins for extended Allen relation predicates. *The VLDB Journal* 30, 3 (Feb. 2021), 379–402. doi:10.1007/s00778-020-00650-5
- [40] Danila Piatov, Sven Helmer, and Anton Dignös. 2016. An interval join optimized for modern hardware. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 1098–1109. doi:10.1109/ICDE.2016.7498316
- [41] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2015. Scaling up concurrent main-memory column-store scans: towards adaptive NUMA-aware data and task placement. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1442–1453. doi:10.14778/2824032.2824043
- [42] Yeasir Rayhan and Walid G. Aref. 2023. SIMD-ified R-tree Query Processing and Optimization. In *Proceedings of the 31st ACM International Conference on Advances in Geographic Information Systems* (Hamburg, Germany) (SIGSPATIAL '23). Association for Computing Machinery, Article 37, 10 pages.
- [43] Ioannis Reppas, Meghdad Mirabi, Leila Fathi, Carsten Binnig, Anton Dignös, and Johann Gamper. 2024. Parallel Processing of Temporal Anti-Joins in Memory. In *Database Systems for Advanced Applications: 29th International Conference, DASFAA 2024, Gifu, Japan, July 2-5, 2024, Proceedings, Part I*. Springer-Verlag, Berlin, Heidelberg, 86–102. doi:10.1007/978-981-97-5552-3_6
- [44] A. Segev and H. Gunadhi. 1989. Event-join optimization in temporal relational databases. In *Proceedings of the 15th International Conference on Very Large Data Bases* (Amsterdam, The Netherlands) (VLDB '89). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 205–215.
- [45] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Apusawamy, and Anastasia Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 698–709. doi:10.1109/ICDE.2019.00068
- [46] Utku Sirin and Anastasia Ailamaki. 2020. Micro-architectural analysis of OLAP: limitations and opportunities. *Proc. VLDB Endow.* 13, 6 (Feb. 2020), 840–853. doi:10.14778/3380750.3380755
- [47] Utku Sirin, Pinar Tözün, Danica Porobic, and Anastasia Ailamaki. 2016. Micro-architectural Analysis of In-memory OLTP. 387–402. doi:10.1145/2882903.2882916
- [48] Mike Stonebraker, Daniel Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2018. *C-store: a column-oriented DBMS*. 491–518. doi:10.1145/3226595.3226638
- [49] Jens Teubner and Rene Mueller. 2011. How Soccer Players Would Do Stream Joins. *Proceedings of the ACM SIGMOD International Conference on Management of Data*. doi:10.1145/1989323.1989389
- [50] Pinar Tözün, Brian Gold, and Anastasia Ailamaki. 2013. OLTP in wonderland: where do cache misses come from in major OLTP components? doi:10.1145/2485278.2485286
- [51] Ahmad Yasin. 2014. A Top-Down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 35–44. doi:10.1109/ISPASS.2014.6844459
- [52] Donghui Zhang, Vassilis J. Tsotras, and Bernhard Seeger. 2002. Efficient Temporal Join Processing Using Indices. In *Proceedings of the 18th International Conference on Data Engineering (ICDE '02)*. IEEE Computer Society, USA, 103–113.
- [53] Hao Zhang, Xianzhi Zeng, Shuhao Zhang, Xinyi Liu, Mian Lu, and Zhao Zheng. 2023. Scalable Online Interval Join on Modern Multicore Processors in OpenMLDB. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 3031–3042. doi:10.1109/ICDE55515.2023.00232