An Interval Join Optimized for Modern Hardware

Danila Piatov*, Sven Helmer* and Anton Dignös*
* Faculty of Computer Science
Free University of Bozen-Bolzano, Italy
Email: firstname.lastname@unibz.it

***Abstract*—We develop an algorithm for efficiently joining relations on interval-based attributes with overlap predicates, which, for example, are commonly found in temporal databases. Using a new data structure and a lazy evaluation technique, we are able to achieve impressive performance gains by optimizing memory accesses exploiting features of modern CPU architectures. In an experimental evaluation with real-world datasets our algorithm is able to outperform the state-of-the-art by an order of magnitude.**

I. Introduction

Interval-based data types and the infrastructure for storing and processing them, commonly found in temporal and spatial databases, are spreading into other areas as well, clearly illustrating the need for this kind of functionality. For instance, the SQL:2011 standard now includes temporal features [1], such as the `overlaps` predicate, and there is also work on implementing interval joins on top of map-reduce frameworks [2]. An overlap interval join, the operator we are focusing on, can be used in many different application areas. For instance, assume we want to analyze weather data and are looking for overlapping periods of high temperatures and heavy rainfall [2]. Figure 1 illustrates this by matching time periods of different weather conditions. While t_1 overlaps with r_1 and t_2 with r_3, r_2 does not find a join partner. The benefit of using intervals for modeling data goes beyond temporal events, though. For instance, they can be helpful in managing uncertain data in which attribute values are represented by ranges [3]. Last but not least, utilizing them for filtering potential candidate objects in multi-dimensional data by querying multiple dimensions in the form of intervals or using space-filling curves is also an option [4]. Here we focus on general techniques of processing overlap joins. Nevertheless, our algorithm is also applicable in combination with other joins. Temporal equi-join, for instance, can be implemented by first hashing the two tables and then running an overlap join on the corresponding partitions.

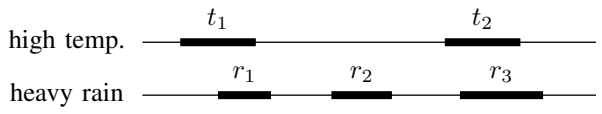


Fig. 1. Matching overlapping time periods

Due to the availability of larger and larger main memory sizes, more and more data is stored in memory, in fact main memory database systems are becoming ever more popular [5], [6], [7]. As a consequence, the implementation of basic operators and whole query engines in these systems has to be adapted to the modern hardware. The gap in efficiency in the storage hierarchy between CPU caches and main memory has become as important as, if not more important than, the gap between main memory and disk. In fact, since memory access times have not been able to keep up with the increasing speed of CPUs, accessing memory has become a major bottleneck, often forcing CPUs to wait for data to process. This has led to the development of cache-conscious and cache-oblivious data structures and algorithms for database systems to increase performance, see [8], [9], [10], [11] for some examples.

By applying techniques that keep the memory latency low to a variant of a sweep-based interval join, which has been neglected due to performance reasons so far, we show that this algorithm is actually able to outperform the state of the art and maintain its good performance on a very wide range of workloads. In particular, we make the following contributions:

- We introduce a sweep-based overlap interval join algorithm together with a novel compact hash table and lazy processing technique.
- We analyze our data structure and algorithm theoretically, illustrating where their main strengths lie.
- We experimentally evaluate our algorithm and demonstrate that it outperforms the state-of-the-art partitioning algorithm by up to a factor of six and another sweep line algorithm by up to a factor of nine on real-world data.

The remainder of the paper is organized as follows. We summarize related work in Section II, followed by a formal definition of interval joins in Section III. In Section IV we introduce our Endpoint-Based Interval Join and in Section VI we show how to implement it efficiently, after reviewing the features of modern hardware architectures in Section V. Section VII contains an analysis of the efficiency of our algorithm, followed by an empirical evaluation in Section VIII. We conclude with a summary and an outlook in Section IX.

II. Related Work

One of the earliest publications to look at performance issues of temporal joins is by Segev and Gunadhi [12], [13], who compare different sort-merge and nested-loop implementations of their event join. They refined existing algorithms by applying an auxiliary access method called an append-only tree, assuming that temporal data is only appended to existing relations and never updated or deleted. Leung and Muntz [14] and Soo et al. [15] introduced partitioning to temporal joins, showing that it compared favorably to nested-loop and sort-merge join algorithms.

Some of the work on spatial joins can be applied to interval joins as well. Arge et al. [16] used a sweeping-based interval

join algorithm as a building block for a two-dimensional spatial rectangle join, but did not investigate it as a standalone interval join. It was picked up again by Gao et al. [17], who give a taxonomy of temporal join operators and provide a survey and empirical study of a considerable number of non-index-based temporal join algorithms, in particular variants of nested-loop, sort-merge, and partitioning-based methods. Again, the sort-merge variants cannot keep up with the partition-based approaches. The studies were conducted on a simulation testbed with fixed parameters for I/O and other costs, though, and in terms of caching a simplified model was used, assuming a high cache hit ratio (>90%). In particular, none of the algorithms was modified to make use of the cache explicitly.

There are several techniques based on supporting the join processing with index structures, such as segment trees [18], interval trees [19], relational interval trees [20], and quadtrees [21]. The Timeline Index [22] used in the SAP HANA research prototype is similar to the index structure we use. However, the HANA access method is more complex, as it is space-optimized for transaction-time temporal databases. For our purposes the event list structure of a Timeline Index is sufficient. The Timeline Join algorithm implemented on top of the Timeline Index is very similar to a naïve version of our basic join algorithm and does not optimize memory accesses.

The most recent work on interval joins, the overlap interval partitioning (OIP) join, which we consider to be the state-of-the-art approach, was developed by Dignös et al. [23]. The (temporal) domain is divided into equally-sized granules and adjacent granules can be combined to form containers of different sizes. Intervals are assigned to the smallest container that covers them and the join algorithm then matches intervals in overlapping containers. In an experimental evaluation this algorithm comes out on top against sort-merge and index-based approaches. Nevertheless, the sort-merge variant turns out to be more efficient for very short intervals.

There is a substantial body of work on adapting hashing and hash-join algorithms to modern hardware. A prominent example is the radix hash-join [11], [24]. However, the focus of our work is completely different. For a (radix) hash-join entire relations are partitioned and then remain unchanged during the probing phase. Consequently, the goal here is to minimize TLB misses during the partitioning phase. In contrast, we have a hash table that constantly changes while processing a join and we need to minimize the costs for scanning the whole hash table.

III. Definition of Interval Joins

Interval data appears in many application domains, but arguably the most common is temporal data where intervals denote time periods. Without loss of generality we use temporal intervals and terminology from the area of temporal databases to define our interval join. A temporal interval join is also known by other names, such as *time join* (*T-join*) [13], [12] or *temporal Cartesian product* [17].

Definition 1: An *interval relation* $\mathbf{r}$ is a relation with tuples $r = (A_1, \ldots, A_n, T_s, T_e)$ where T_s and T_e denote, respectively, left and right endpoints of a closed interval[1], and A_i, $1 \leq i \leq n$ are other attributes (called *explicit attributes* in temporal databases).

The interval denoted by the two endpoints can be written as $[T_s, T_e]$ or simply as T. We will use a period (.) to reference an attribute of a particular tuple: $r_i.T$ or $r_i.T_s$. We use closed intervals, but our approach can be easily extended to open and half-open intervals (as used, e.g. in SQL:2011).

Definition 2: An *interval join* of two interval relations $\mathbf{r}$ and $\mathbf{s}$ is a relational join with a predicate checking that tuple intervals overlap. More formally, $r.T$ intersects with $s.T$, i.e., $r.T \cap s.T \neq \emptyset$. This inequality can also be rewritten as $r.T_s \leq s.T_e \wedge s.T_s \leq r.T_e$.

So an interval join can be seen as a join using a conjunction of two inequality predicates. Consequently, it is more difficult to compute than a regular equi-join.

Example 1: Consider interval relations $\mathbf{r}$ and $\mathbf{s}$ in Figure 2. Relation $\mathbf{r}$ consists of three tuples: $r_1.T = [1, 5]$, $r_2.T = [1, 10]$, and $r_3.T = [7, 11]$. Relation $\mathbf{s}$ consists of five tuples: $s_1.T = [2, 2]$, $s_2.T = [3, 12]$, $s_3.T = [4, 5]$, $s_4.T = [5, 6]$, $s_5.T = [8, 9]$.

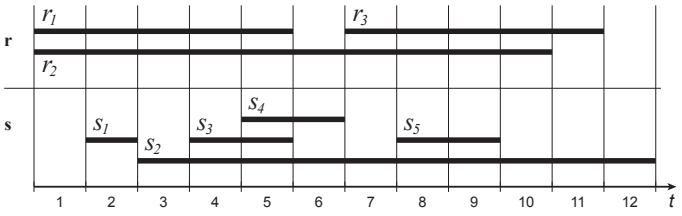


Fig. 2. Example interval relations

The result of the interval join between the two relations is the following list of output pairs: $\langle r_1, s_1\rangle$, $\langle r_1, s_2\rangle$, $\langle r_1, s_3\rangle$, $\langle r_1, s_4\rangle$, $\langle r_2, s_1\rangle$, $\langle r_2, s_2\rangle$, $\langle r_2, s_3\rangle$, $\langle r_2, s_4\rangle$, $\langle r_2, s_5\rangle$, $\langle r_3, s_2\rangle$, $\langle r_3, s_5\rangle$.

IV. Endpoint-Based Interval Join

In this section we present the basic version of our *endpoint-based interval join* algorithm (EBI-Join). It is loosely based on an internal-memory plane-sweep technique [25] and has similarities with the interval join by Arge et al. [16]. A slightly different version of EBI-Join is used as part of the temporal join in HANA [22]. First, we briefly describe the index and then go on to the actual join algorithm.

A. Endpoint Index

The idea of the endpoint index is that intervals, which can be seen as points in a two-dimensional space, are mapped onto one-dimensional endpoints or *events*. Let $\mathbf{r}$ be an interval relation with tuples r_i, $1 \leq i \leq n$. A tuple r_i in an endpoint index is represented by two events of the form $e = \langle timestamp, type, tuple\_id\rangle$, where $timestamp$ is T_s or T_e of the tuple, $type$ is either start or end and $tuple\_id$ is the tuple identifier, i.e., the two events for a tuple r_i are $\langle r_i.T_s, \mathrm{start}, i\rangle$ and $\langle r_i.T_e, \mathrm{end}, i\rangle$. For instance, for $r_3.T = [7, 11]$, the two events are $\langle 7, \mathrm{start}, 3\rangle$ and $\langle 11, \mathrm{end}, 3\rangle$, which can be seen as "at time 7 tuple 3 started" and "at time 11 tuple 3 ended".

[1] In the following we use "left" and "start" interchangeably and also "right" and "end".

Since events represent timestamps (and not intervals) of tuples, we can impose a total order among events, where the order is according to $timestamp$ and ties are broken by $type$. From our definition of closed intervals (see Def. 1) it follows that a starting event precedes an ending event with the same timestamp (i.e. start $<$ end). In case of open intervals, the ordering of types would be reversed. Endpoints with equal timestamps and types but different tuple identifiers are considered equal. An endpoint index for interval relation $\mathbf{r}$ is the ordered list of events $[e_1, e_2, \ldots, e_{2n}]$ sorted in ascending order.

To build the endpoint index for a relation we need to sort the endpoints of all tuples according to their total order. The index will have twice the number of elements compared to the tuple count in the relation (an individual element of the index is very compact, though), which still gives us $O(n)$ space complexity for storing it and $O(n \log n)$ time complexity for building.

Example 2: Consider interval relations $\mathbf{r}$ and $\mathbf{s}$ in Figure 2. The endpoint index for relation $\mathbf{r}$ is $[\langle 1, \text{start}, 1\rangle, \langle 1, \text{start}, 2\rangle, \langle 5, \text{end}, 1\rangle, \langle 7, \text{start}, 3\rangle, \langle 10, \text{end}, 2\rangle, \langle 11, \text{end}, 3\rangle]$. The endpoint index for relation $\mathbf{s}$ is $[\langle 2, \text{start}, 1\rangle, \langle 2, \text{end}, 1\rangle, \langle 3, \text{start}, 2\rangle, \langle 4, \text{start}, 3\rangle, \langle 5, \text{start}, 4\rangle, \langle 5, \text{end}, 3\rangle, \langle 6, \text{end}, 4\rangle, \langle 8, \text{start}, 5\rangle, \langle 9, \text{end}, 5\rangle, \langle 12, \text{end}, 2\rangle]$.

B. Basic Join Algorithm

We scan the endpoint indexes of two argument relations in an interleaved fashion, keeping track of intervals that have started but not yet finished. Encountering a left endpoint during the scan means that the interval of a tuple has started, encountering a right endpoint means the interval has ended. We call tuples that have started but not yet finished *active* tuples. When a tuple from one relation starts, it is added to the set of active tuples for that relation. When it finishes, we remove it from the set. Active tuples are maintained by in-memory maps (associative arrays) of tuple identifiers to tuples. When an interval starts, it produces a Cartesian product with the active tuples from the other relation as the output of the algorithm. See Algorithm 1 (EBI-Join) for the pseudocode.

We prove the algorithm is correct by observing that two tuples from two different relations can be in the sets of active tuples at the same time if and only if they overlap. The algorithm outputs such a pair only once—when the latter of the two intervals is added to the corresponding active set. More formally, EBI-Join is a sweep algorithm that traverses the timeline based on a list of events, in our case the left and right endpoints of all intervals. The transition procedure for each event is either an insert operation followed by the generation of output tuples or a remove operation. The following invariant is maintained after processing a given event e. For each tuple r (s) in active set $active^r$ ($active^s$) it holds that $r.T_s \leq e$ and $r.T_e \geq e$ ($s.T_s \leq e$ and $s.T_e \geq e$), from which follows $r.T_s \leq s.T_e \wedge s.T_s \leq r.T_e$, i.e., the intervals overlap.

Example 3: We show the mode of operation of EBI-Join using the example data from Figure 2 and the endpoint indexes from Example 2. The algorithm starts scanning the indexes, encounters first the left endpoints of intervals r_1 and r_2, loads these tuples from relation r, and adds them to $active^r$. It then encounters the left endpoint of s_1, loads the tuple, adds it to $active^s$, and produces the Cartesian product between s_1 and the tuples in $active^r$, namely r_1 and r_2. When encountering the right endpoint of s_1 next, we remove tuple s_1 from $active^s$. Encountering the left endpoint of s_2 repeats the process: add s_2 to $active^s$ and generate output tuples with the tuples in $active^r$. The algorithm continues until we reach the end of one of the indexes (in our example this is the index of r), producing the following output: $\langle r_1, s_1\rangle$, $\langle r_2, s_1\rangle$, $\langle r_1, s_2\rangle$, $\langle r_2, s_2\rangle$, $\langle r_1, s_3\rangle$, $\langle r_2, s_3\rangle$, $\langle r_1, s_4\rangle$, $\langle r_2, s_4\rangle$, $\langle r_3, s_2\rangle$, $\langle r_2, s_5\rangle$, $\langle r_3, s_5\rangle$.

Algorithm 1: Endpoint-Based Interval (EBI) Join

```
input  : Interval relations r and s, endpoint indexes e^r and e^s
output : Joined tuple pairs
1  active^r ← new Map of tuple identifiers to tuples
2  active^s ← new Map of tuple identifiers to tuples
3  e^r ← first(e^r)
4  e^s ← first(e^s)
5  while exists(e^r) and exists(e^s) do
6      if e^r < e^s then
7          if e^r.type = start then
8              r ← r[e^r.tuple_id]
9              active^r[e^r.tuple_id] ← r
10             for s ∈ active^s do
11                 output(r, s)
12             end
13         else
14             active^r.remove(e^r.tuple_id)
15         end
16         advance(e^r)
17     else
18         if e^s.type = start then
19             s ← s[e^s.tuple_id]
20             active^s[e^s.tuple_id] ← s
21             for r ∈ active^r do
22                 output(r, s)
23             end
24         else
25             active^s.remove(e^s.tuple_id)
26         end
27         advance(e^s)
28     end
29 end
```

C. Managing the Active Tuple Set

For managing the active tuple set we need a data structure into which we can `insert` key-value pairs, `remove` them, and quickly enumerate (scan) one by one all the values contained in the data structure via the operation `getnext`. In our case, the keys are tuple identifiers and the values are the tuples themselves. The data structure of choice here is a map or associative array.

The most efficient implementation of a map optimizing the `insert` and `remove` operations is a hash table (with $O(1)$ time complexities for these operations). However, hash tables are not well-suited for scanning. The *std::unordered_map* class in the C++ Standard Template Library and the *java.util.HashMap* in the Java Class Library, for instance, scan through all the buckets of a hash table, making the performance of a scan operation linear with respect to the capacity of the hash table and not to the actual amount of elements in it.

In order to achieve an $O(1)$ complexity for `getnext`, the elements in the hash table can be connected via a doubly-linked list (see Figure 3). The hash table stores pointers to elements, which in turn contain a key, a value, two pointers for the doubly-linked list (*list prev* and *list next*) and a pointer for chaining elements of the same bucket for collision resolution (pointer *bucket next*). This approach is employed in the *java.util.LinkedHashMap* in the Java Class Library.

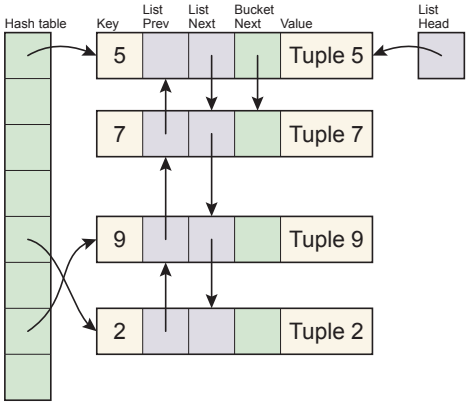


Fig. 3. Linked hash map

While this data structure offers a constant complexity for `getnext`, the execution times of different calls of `getnext` can vary widely in practice, depending on the memory footprint of the map. After a series of insertions and deletions the elements of the linked list become randomly scattered in memory, which results in slow random memory accesses for retrieving list elements. However, for our approach it is crucial that `getnext` can be executed very efficiently, as it is typically called much more often than `insert` and `remove`.

D. Parallel Execution

The EBI-Join can also be easily parallelized. Both input relations $\mathbf{r}$ and $\mathbf{s}$ are virtually divided into a number of equally-sized partitions. First, the tuples of a relation are sorted by their starting time and then assigned to the partitions in a round-robin fashion, i.e., the i-th tuple is assigned to partition $(i \mod k)$, where k is the number of partitions. A separate endpoint index is then built for each partition. This method keeps the size of the active tuple sets small, as close neighbors are assigned to different partitions. This is very important, because the cardinality of active tuple sets has a major impact on the run time of the algorithm (see Section VII-B4). Second, we do pairwise joins between all partitions of $\mathbf{r}$ with all partitions of $\mathbf{s}$. Since the partitions of a relation are disjoint, we can run all these joins independently of each other and we also do not need a merging step in the end. In this way, we can make good use of multiple CPU cores in a system.

V. Features of Modern Hardware

In the following we briefly review mechanisms employed by modern hardware to decrease main memory latency. This latency can have a huge impact, as fetching data from main memory may easily use up more than a hundred CPU cycles.

A. Mechanisms

Usually, there is a hierarchy of caches, with smaller, faster ones closer to CPU registers. Cache memory has a far lower latency than main memory, so a CPU first checks whether the requested data is already in one of the caches (starting with the L1 cache, working down the hierarchy). Not finding data in a cache is called a *cache miss* and only in the case of cache misses on all levels, main memory is accessed. In practice an algorithm with a small memory footprint runs much quicker, because in the ideal case, when an algorithm's data (and code) fits into the cache, the main memory only has to be accessed once at the very beginning, loading the data (and code) into the cache.

Besides the size of a memory footprint, the access pattern also plays a crucial role, as modern hardware contains *prefetchers* that speculate on which blocks of memory will be needed next and preemptively load them into the cache. The easier the access pattern can be recognized by a prefetcher, the more effective it becomes. Sequential access is a pattern that can be picked up by prefetchers very easily, while random access effectively renders them useless.

Also, programs do not access physical memory directly, but through a virtual memory manager, i.e., virtual addresses have to be mapped to physical ones. Part of the mapping table is cached in a so-called *translation lookaside buffer* (TLB). As the size of the TLB is limited, a program with a high level of locality will run faster, as all look-ups can be served by the TLB.

Out-of-order execution (also called *dynamic execution*) allows a CPU to deviate from the original order of the instructions and run them as the data they process becomes available. Clearly, this can only be done when the instructions are independent of each other and can be run concurrently without changing the program logic.

Finally, certain properties of DRAM (dynamic random access memory) chips also influence latency. Accessing memory using fast page or a similar mode means accessing data stored within the same page or bank without incurring the overhead of selecting it. This mechanism favors memory accesses with a high level of locality.

B. Performance Numbers

We provide some numbers to give an impression of the performance of currently used hardware. For contemporary processors, such as "Core" and "Xeon" by Intel[2], one random memory access within the L1 data (*L1d*) cache (32 KB per core) takes 4 CPU cycles. Within the L2 cache (256 KB per core) one random memory access takes 11–12 cycles. Within the L3 cache (3–45 MB) one random memory access takes 30–40 CPU cycles. Finally, one random physical RAM access takes around 70–100 ns (200–300 processor cycles). It follows that the performance gap between an L1 cache access and a main memory access is huge: two orders of magnitude.

[2]We use the cache and memory latencies obtained for the Sandy Bridge family of Intel CPUs using the SiSoftware Sandra benchmark, `http://www.sisoftware.net/?d=qa&f=ben_mem_latency`.

VI. Improving Memory Access

Managing the active tuple set efficiently in terms of memory accesses is crucial for the performance of the join algorithm. In the following we show how to improve the basic algorithm considering the features of current hardware.

A. Optimizing Access to the Active Tuple Set

In the experimental section (Section VIII) we show that the basic join algorithm from Section IV basically starves the CPU. Our goals have to be to store the active tuple set as compactly as possible and to access it sequentially, allowing the hardware to get the data to the CPU in an efficient manner.

We store the elements of our hash map in a contiguous memory area. For the `insert` operation this means that we always append a new element at the end of the storage area. Removing the last element from the storage area is straightforward. If the element to be removed is not the last in the storage area, we swap it with the last element and then remove it. When doing so, we have to update all the references to the swapped elements. Scanning involves stepping through the contiguous storage area sequentially. We call our data structure a *gapless hash map* (see Figure 4).

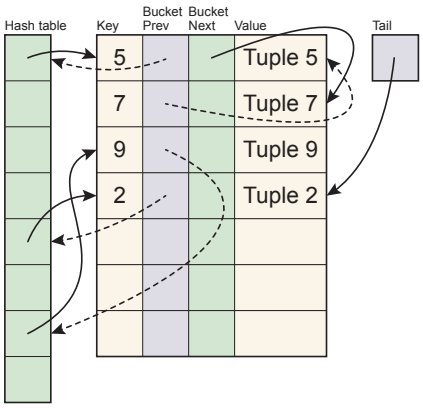


Fig. 4. Gapless hash map

The hash table stores pointers to elements, which contain a key, a value, a pointer for chaining elements of the same bucket when resolving collisions (pointer *bucket next*, solid arrows), and a pointer *bucket prev* to a hash table entry or an element (whichever holds the forward pointer to this element, dashed arrows). The latter is used for updating the reference to an element when changing the element position. The main difference to the random memory access of a linked hash map (Fig. 3) is the allocation of all elements in a contiguous memory area, allowing for fast sequential memory access when enumerating the values.

Example 4: Assume we want to remove tuple 7 from the structure depicted in Figure 4. First of all, the bucket-next pointer of the element with key 5 is set to NULL. Next, the last element in the storage area (tuple 2) is moved to the position of the element with key 7. Following the bucket-prev pointer of the just moved element we find the reference to the element in the hash table and update it. Finally, the variable *tail* is decremented to point to the element with key 9.

We can go even further by separating the tuples from the elements, storing them in a different contiguous memory area in corresponding locations. All basic element operations (append and move) are mirrored for the corresponding tuples. This slightly increases the costs for insertions and removal of tuples. However, scanning the tuples is as fast as it can become, because we do not need to read any metadata, only tuple information.

B. Lazy Joining of the Active Tuple Set

The fastest `getnext` operations are those that are not executed. We modify our algorithm to boost its performance by significantly reducing the number of `getnext` operations needed to generate the output.

We illustrate our point using the example setting in Figure 2. Assume we have just encountered the left endpoint of s_1, which means that our algorithm now scans the tuple set $active^r$, which contains r_1 and r_2. After that we scan it again and again when encountering the left endpoints of s_2, s_3, and s_4. However, since no endpoints of $\mathbf{r}$ were encountered during that time, we scan the same version of $active^r$ four times. We can reduce this to one scan if we keep track of the tuples s_1, s_2, s_3, and s_4 in a (contiguous) buffer, delaying the scan until there is about to be a change in $active^r$.

This is the main idea of the optimized version of our algorithm: when there is a sequence of intervals starting in one relation with no start or end events in the other relation, we only execute a single scan. The pseudocode for our Lazy Endpoint-Based Interval (LEBI) Join can be found in Algorithm 2. The capacity c of the buffer should be chosen in such a way that it fits into the L1d CPU cache (line 5). This will make buffer accesses significantly faster than scanning active tuple sets.

Example 5: Assume we are at the beginning of the third iteration of the main loop (line 6). We have already encountered the left endpoints of r_1 and r_2 and the current state is the following: $active^r = \{r_1, r_2\}$, $active^s = \emptyset$, and the events next in line for $\mathbf{r}$ and $\mathbf{s}$ are $e^r = \langle 5, \text{end}, 1\rangle$ and $e^s = \langle 2, \text{start}, 1\rangle$, respectively. The buffer was emptied at the end of the previous iteration of the loop (line 44).

The next event that is processed is e^s (determined in line 7), branching to the *else* clause (line 25). Lines 26–35 process the index of $\mathbf{s}$, collecting the starting intervals of s_1, s_2, s_3, and s_4 in the buffer (as nothing happens in $\mathbf{r}$). While doing so, the algorithm also keeps $active^s$ up to date as usual. Here the loop will execute six times, with $active^s = \{s_2, s_4\}$ and having collected $\{s_1, s_2, s_3, s_4\}$ in the buffer. Next, since the buffer is not empty, $active^r$ is scanned once, producing the Cartesian product of $\{r_1, r_2\}$ with $\{s_1, s_2, s_3, s_4\}$.

VII. Efficiency Analysis

A. Cost Model

The total cost $c_{r \bowtie s}$ for joining the relations $\mathbf{r}$ and $\mathbf{s}$ (having cardinalities n_r and n_s, respectively) comprises the cost c_r for processing the tuples in $\mathbf{r}$, the cost c_s for processing the tuples in $\mathbf{s}$, and the cost c_z for generating the output tuples, so $c_{r \bowtie s} = c_r + c_s + c_z$. Processing tuples from the input relation $\mathbf{r}$ means we `read` two endpoint events from the index, `load` the tuple from the relation, and `insert` it into and `remove` it from the active tuple set. (the same holds for the relation $\mathbf{s}$). Therefore,

$$c_r = n_r(2 \cdot c_{read} + c_{load} + c_{insert} + c_{remove}).$$

Algorithm 2: Lazy Endpoint-Based Interval (LEBI) Join

```
input  : Interval relations r and s, endpoint indexes e^r and e^s
output : Joined tuple pairs
active^r ← new Map of tuple identifiers to tuples
active^s ← new Map of tuple identifiers to tuples
e^r ← first(e^r)
e^s ← first(e^s)
buf ← new resizable array of tuples with capacity c
while exists(e^r) and exists(e^s) do
    if e^r < e^s then
        repeat
            if e^r.type = start then
                r ← r[e^r.tuple_id]
                active^r[e^r.tuple_id] ← r
                buf.insert(r)
            else
                active^r.remove(e^r.tuple_id)
            end
            advance(e^r)
        until not exists(e^r) or e^s < e^r or buf.isFull
        if buf.isNotEmpty then
            for s ∈ active^s do
                for r ∈ buf do
                    output(r, s)
                end
            end
        end
    else
        repeat
            if e^s.type = start then
                s ← s[e^s.tuple_id]
                active^s[e^s.tuple_id] ← s
                buf.insert(s)
            else
                active^s.remove(e^s.tuple_id)
            end
            advance(e^s)
        until not exists(e^s) or e^r < e^s or buf.isFull
        if buf.isNotEmpty then
            for r ∈ active^r do
                for s ∈ buf do
                    output(r, s)
                end
            end
        end
    end
    buf.clear
end
```

Here, c_{read} represents reading an index element, which is always sequential and therefore fast even in case of external memory, while the cost of loading a tuple (c_{load}) is one random memory access. It is possible to improve this by sorting the relation by T_s and ensuring that tuples with the same starting time are in the same order as their endpoints in the index. In that case the algorithm will read each relation only once sequentially.

Generating the output tuples (in case of EBI-Join) requires scanning active tuple sets, fetching tuples one by one using `getnext`, so

$$c_z = n_z \cdot c_{getnext}.$$

where n_z is the cardinality of the output relation. Next, we compare c_{insert}, c_{remove} and $c_{getnext}$ of a linked hash map to a gapless hash map.

B. Data Structures

In our implementation we use fixed-size hash tables. We estimate the size by either building the indexes on the fly or by keeping statistics on them (which have to be updated from time to time). The size of the hash table is chosen as the first power of two greater or equal to the maximum active tuple count (thus keeping the maximum load factor α between 0.5 and 1). The average bucket size in the case of hashing with chaining is $1 + \alpha$, and since all our searches are successful, the average number of comparisons we have to make when searching for a tuple is $1 + \alpha/2$, which is between 1.25 and 1.5. Due to the changing content of the hash table, the average number of active tuples for a relation is usually lower than the maximum, so the actual average load factor will be even smaller, resulting in a smaller average number of comparisons.

We express the costs in terms of random and sequential memory accesses. One *rndA* ("random access") means that an operation accesses (for reading and/or writing) a random location within a memory area. This can cause costly delays if the memory area is too large to fit into the L1d CPU cache. One *seqA* ("sequential access") means that a data item is read from or written to a memory area during a sequential scan or accessed repeatedly within a short time frame. This operation causes almost no delays.

1) Linked Hash Map: Inserting a tuple into an active tuple set using a linked hash map (Fig. 3) boils down to the following: (1) insert a new tuple at the head of the list by allocating a new list element (1 rndA of available element stack), copying the tuple into it (1 rndA + 1 seqA), and setting the correct ListNext value, (2) modify the ListPrev pointer of the previous first element (1 rndA), (3) update the directory entry (1 rndA), and (4) adjust the list head (we assume that it stays in cache). If we have a collision in the hash table, then in step (1) we will also have to modify the BucketNext pointer of the newly inserted element, placing it first in the bucket.

$$c_{insert}^{linkedhashmap} = 4 \text{ rndA} + 1 \text{ seqA}.$$

When removing a tuple, we have to go through the following steps: (1) look up the tuple in the directory and update the directory (1 rndA), (2) look up and modify the ListNext pointer of the previous element in the list (1 rndA), (3) look up and modify the ListPrev pointer of the next element (1 rndA), and (4) free the element (1 rndA of available element stack). (From time to time, we also have to update the list head pointer.) In case of a collision, we have to find the needed element in the bucket and update the BucketNext pointer of the previous bucket element, where each step results in a random memory access. However, as we mentioned above, the collision rate in our case is negligibly small.

$$c_{remove}^{linkedhashmap} = 4 \text{ rndA}.$$

When fetching a tuple to produce output, calling `getnext` will follow the ListNext pointer of the current item. Due to elements being randomly scattered in the memory caused by the constant insertion and removal of items, every call of `getnext` costs one random access and one sequential tuple read.

$$c_{getnext}^{linkedhashmap} = 1 \text{ rndA} + 1 \text{ seqA}.$$

2) Gapless Hash Map: When inserting a tuple into an active tuple set using our gapless hash map (Fig. 4), we apply these steps: (1) copy the new element into the tail of the element storage (1 rndA + 1 seqA), (2) update the directory entry (1 rndA), and (3) adjust the Tail pointer. In case of a collision, we have an additional step: (4) modify the BucketPrev pointer of the previous first element in the bucket..

$$c_{insert}^{gaplesshashmap} = 3 \text{ rndA} + 1 \text{ seqA}.$$

When removing a tuple, we have to do the following steps: (1) look up the tuple identifier in the directory, (2) fetch the last entry from the hash table, (3) copy it to the position of the tuple that is to be removed (copying costs 2 seqA), (4) adjust the Tail pointer, and (5) update the directory entry. If the removed tuple is part of a bucket, we have to (6) find it in the bucket, (7) adjust the BucketPrev pointer of the next element and (8) the BucketNext pointer of the previous element. If the moved, last tuple is part of a bucket, we will also have to do steps (7) and (8) for that tuple as well. Here, all steps except (4) and (8) can result in cache misses.

$$c_{remove}^{gaplesshashmap} = 5 \text{ rndA} + 2 \text{ seqA}.$$

Generating output tuples is done by sequentially scanning a contiguous area of memory:

$$c_{getnext}^{gaplesshashmap} = 1 \text{ seqA}.$$

3) Comparison: When computing the EBI-Join, every tuple of the input relations is inserted into and removed from the hash map exactly once. Compared to a linked hash map, the gapless hash map has an additional two $seqA$ per input tuple. The `getnext` function, on the other hand, is called once for each output tuple. Here, the linked hash map is more expensive: it needs an additional $rndA$. We now take a look at what difference two $seqA$ versus one $rndA$ make in practice.

4) Experimental Evaluation: We filled the hash maps with various numbers of 32-byte tuples, then randomly added and removed tuples to simulate the management of an active tuple set. We can execute a sequential access in at most 3–4 ns, which means that the difference between the linked and the gapless hash map is a few nanoseconds for the two $seqA$.

The differences for the single $rndA$ can be huge, though. Figure 5, shows the average latency of a `getnext` operation depending on the size of the map (note the logarithmic scale).

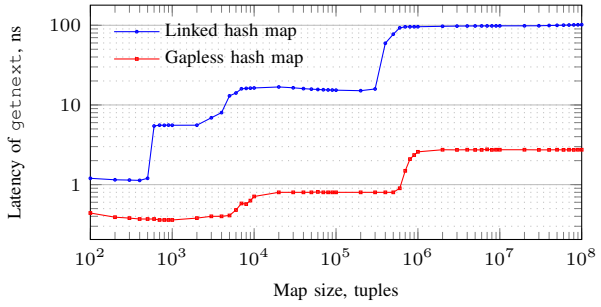


Fig. 5. Latency of `getnext` operation

We see that the latency of a `getnext` operation is not constant but grows depending on the memory footprint of the tuples. In order to find the cause of this, we used the Performance Application Programming Interface (PAPI) library to read out the CPU performance counters [26]. When looking at the average number of stalled CPU cycles (PAPI-RES-STL) per `getnext` operation, we get a very similar picture (see Figure 6). Therefore, the latency is obviously caused by the CPU memory subsystem.

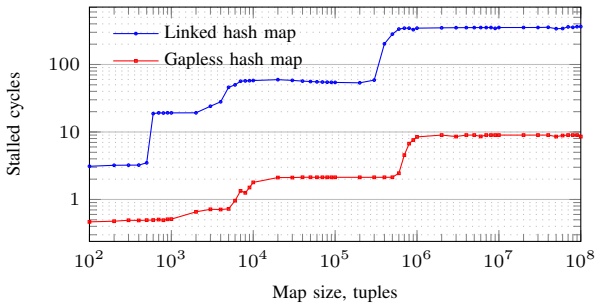


Fig. 6. Stalled CPU cycles per `getnext` operation

In Figure 5 we can clearly identify three distinct transitions. When we have a small number of tuples, all of them fit into the L1d CPU cache (32 KB) and we have a low latency. As the tuple count grows towards 1000 tuples, we start using the L2 cache (256 KB), which has a greater latency. When we increase the number of tuples further and start reaching 10,000 tuples, the data is mostly held in the L3 cache (20 MB in our case) and, finally, after arriving at a tuple count of around 600 000, the tuples are mostly located in RAM. We make a couple of important observations. First, due to the more compact storage scheme of the gapless hash map, the transitions set in later. Second, the improvement gains of the gapless hash map are considerable and can be measured in orders of magnitude (note the logarithmic scale). Third, the latency of a `getnext` operation for the gapless hash map plateaus at around 2.7 ns, while the latency for the linked hash map reaches 100 ns.

Cache misses alone do not explain all the latency. Figure 7 shows the average number of cache misses for the L1d (PAPI-L1-DCM), the L2 (PAPI-L2-TCM), and the L3 cache (PAPI-L3-TCM). While in general the average number of cache misses per `getnext` operation is lower for the gapless hash map, the factor between the two hash maps in terms of stalled CPU cycles is disproportionately higher (please note the double-logarithmic scale in Figure 6). Also, the cache misses do not explain the left-most part of Figure 6, in which there are no cache misses at all. The additional performance boost stems from out-of-order execution. Examining the different (slightly simplified) versions of the machine code generated for `getnext` makes this clear. For the gapless hash map, the code looks like this:

```
loop:
  add   rax, [rdx]
  add   rdx, 32          ;increment pointer
  cmp   rcx, rdx
  jne   loop
```

while for the linked hash map we have the following picture:

```
loop:
  add   rax, [rdx]
  mov   rdx, [rdx + 32] ;dereference pointer
  cmp   rcx, rdx
  jne   loop
```

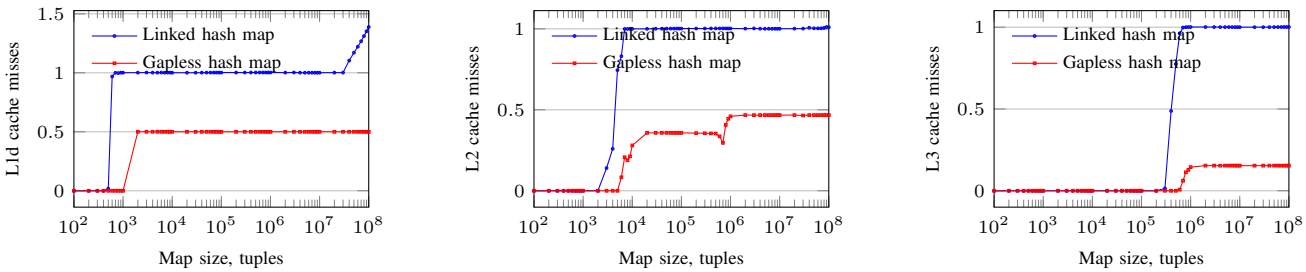


Fig. 7. Cache misses per `getnext` operation

When scanning through a gapless hash map, we add a constant to the pointer, which means that the individual instructions are basically independent of each other. Consequently, the CPU is able to predict the instructions that will be executed next and can already start preparing them out-of-order (i.e., issue cache misses up front for the referenced data) while some of the instructions are still waiting for data from the L1 cache. For the linked hash map the CPU has to wait until the pointer to the next item has been dereferenced. In summary, multiple parallel cache misses in a sequential access pattern are processed much faster than isolated requests to random memory locations.

We made a few further observations: there were no L1 instruction (L1i) cache misses, neither for the gapless nor for the linked hash map. The increase of L1d cache misses for the linked hash map for large numbers of tuples is caused by TLB cache misses. We got very similar results for different CPUs on different machines (the diagrams shown here are for an Intel Xeon E5-2667 v3 processor), which led us to the conclusion that the techniques we employ will generally improve the performance on CPU architectures with a cache hierarchy, prefetching, and out-of-order execution. For the remainder of the paper we only consider the gapless hash map.

VIII. Empirical Evaluation

In this section we provide the results of an empirical study of EBI-Join and LEBI-Join, comparing them with Arge's interval join and overlap interval partition join.

A. Environment

All algorithms were implemented in C++11 by the same author and were compiled with GCC 4.9.2 using `-O3` and `-march=native` optimizations flags to native 64-bit binaries. The execution was performed on a machine with two Intel Xeon E5-2667 v3 processors[3] under Linux. All experiments used 32-byte tuples containing two 64-bit timestamp attributes (T_s and T_e). The experiments were also repeated on a six-year-old Intel Xeon X5550 processor and on a notebook processor i5-4258U, showing a similar behavior.

Common optimizations used in all implementations include: variable (including tuple fields) alignment in memory; using preallocated array-based pools for structure allocation and deallocation (e.g. for linked list elements) instead of using dynamic memory routines, not using virtual function calls, using templates and function inlining (including sorting using a `std::sort` function template with comparison functions inlined into the sorting routine), and loop unrolling. Tuples were implemented as structures, relations were implemented as arrays of tuples. We used the gapless hash map with a separated tuple storage area. The workload consisted of accumulating the sum of XOR operations between the T_s attributes of the joined tuples.

[3]Launched in Q3'14, 8 cores, clock speed 3.2 GHz, max turbo frequency 3.6 GHz, 32 KB per-core L1d cache, 256 KB per-core L2 cache, 20 MB shared L3 cache

B. Competitors

1) Arge's Interval Join: Arge's interval join [16] was originally created as a building block for a spatial rectangle intersection join and is based on in-memory plane-sweeping, making it similar to our EBI-Join. We include it in our study, as to the best of our knowledge it has never been considered in any empirical evaluation of interval joins.

Instead of traversing an endpoint index, it directly scans the interval relations, which have to be sorted on T_s. When a new tuple is encountered, it is placed in a list of active tuples for the scanned relation and we generate output tuples by combining it with the active tuples of the other relation. Outdated tuples are detected and removed during scanning. Consequently, the active tuple lists have to support insertion, scanning, and removal during scanning (by reference). The straightforward implementation of such a data structure is by means of a singly linked list. Due to the random nature of memory accesses during scanning, this structure suffers from the same performance degradation as a linked hash map, though. We improve the implementation by using a *gapless list*, allowing us to access elements sequentially and to remove elements from the middle of the list.

Similarly to the gapless hash map, elements of a gapless list are stored in a contiguous memory area. New elements are appended to the end of the list. When an element is removed from the middle, the last element is moved to its place. For the experiments we only include the implementation of the gapless list, since it always outperformed the linked list.

2) Overlap Interval Partition Join: Currently, the overlap interval partition join or *OIPJoin* by Dignös et al. [23] is considered to be the state-of-the-art approach for interval joins. The algorithm consists of two parts—partitioning and joining.

The partitioning works as follows: first, we choose a number k (more on this later). The smallest interval covering all the intervals in a relation, called the domain, is split into k equally-sized *granules*. Partitions are formed by combining an arbitrary number of adjacent granules, e.g. granule 1, granules 2 and 3, granules 4 to 10 are all valid partitions. Tuples are assigned to

the smallest partition that covers their interval. The algorithm for building the partitions sorts the tuples in a relation by the indexes of the granule that contains the endpoints of their intervals and is then able to create the partitions with one scan over a relation. Note that only non-empty partitions are actually stored.

The join itself is pretty much a standard partitioning join. For each partition of the outer relation, we execute the join by combining it with the overlapping partitions of the inner relation in a block-nested-loop fashion.[4] During this step we have to check whether two intervals actually overlap.

The trickiest part of the algorithm is to choose the correct number of granules k. If k is too small (resulting in a few, large partitions), the nested-loop join combining two partitions does a lot of extra work, because chances are high that many of the intervals in the partitions do not overlap. If k is too large (resulting in a lot of small partitions), the random I/O costs for accessing all these partitions increase. The authors of [23] present a cost model considering statistical properties of the relations, block size, and a CPU/IO cost ratio and based on this develop an algorithm for optimizing k. We implemented and tuned the OIPJoin algorithm specifically for the machine we ran the experiments on.

C. Test Workloads

We tested the algorithms using synthetic and real-world datasets that are described below.

1) Synthetic Datasets: To show particular performance effects of the algorithms we create synthetic datasets with Zipf-distributed [27] starting points of the intervals in the range of $[1, 10^6]$. The duration of the intervals is exponentially distributed with rate parameter λ, which determines the average duration $1/\lambda$ of the intervals. To perform a join, both relations in an individual workload follow the same distribution, but are generated independently with a different seed. We denote a workload z_i as the join between two relation each with 10^7 tuples and $\lambda = 2 \cdot 10^{-i}$. We generated seven such workloads, from the lightest z_0 ($\sim$5.4 tuples on average are active in every relation, $\sim 1.07 \cdot 10^8$ result tuples) to the heaviest z_6 ($\sim 0.3 \cdot 10^7$ active tuples on average, $\sim 0.6 \cdot 10^{14}$ result tuples).

2) Real-world Datasets: We use four real-world datasets that differ in size and data distribution. The main properties of these datasets are summarized in Table I. The Incumbent ("inc") dataset [28] records the history of employees assigned to projects over a 16 year period at a granularity of days. The Feed ("feed") dataset records the history of measured nutritive values of feeds over a 24 year period at a granularity of days; a measurement remains valid until a new measurement for the same nutritive value and feed becomes available. The Webkit ("web") dataset [29] records the history of files of the SVN repository of the Webkit project over a 11 year period at a granularity of milliseconds. The valid times indicate the periods in which a file did not change. The Flight ("flight") dataset [30] is a collection of international flights for November 2014, start and end of the intervals represent plane departure and arrival times with minute precision. For the experiments we used self-joins of these datasets.

[4]The block-nested-loop is actually an improvement we added, our version of OIPJoin is running faster than the original one.

TABLE I. PROPERTIES OF REAL-WORLD DATASETS

	"inc"	"feed"	"web"	"flight"
Cardinality	83,852	3,697,957	1,213,476	57,585
Distinct points	2689	5584	110,165	10,552
Time range	5895	8610	$\sim 2^{39}$	13,543
Min. duration	1	1	$\sim 2^{10}$	25
Max. duration	574	8589	$\sim 2^{39}$	915
Avg. duration	184	432	$\sim 2^{34}$	148

D. Experiments and Results

First, we illustrate the performance gained by using LEBI-Join rather than EBI-Join. Second, LEBI-Join is compared to state-of-the-art algorithms. In the following the measurements include the time for sorting relations (for Arge's and EBI/LEBI-Join), for building endpoint indexes (for EBI/LEBI-Join), and for partitioning the data (for OIPJoin).

1) EBI-Join vs LEBI-Join: The basic EBI-Join algorithm scans $active^r\ n_s$ times and $active^s\ n_r$ times. However, LEBI-Join reduces these numbers considerably. As long as we only encounter starting tuples in $\mathbf{r}$ and no events in $\mathbf{s}$, we can delay the scanning of $active^s$ (and vice-versa).

Analyzing the Data: We now take a look at how long such uninterrupted sequences of starting events are. Figure 8 shows this data for the table "Incumbent" from the real-world datasets when joining it with itself. On the x-axis we have the length of uninterrupted sequences of starting events and on the y-axis their relative frequency of appearance. In 60% of the cases we have sequences of length ten or more, meaning that our LEBI-Join can avoid a considerable number of scans on active tuple sets.

We found that starting events of intervals are not uniformly distributed in real-world datasets, but tend to cluster around certain time points. This can be recognized by looking at the number of distinct points in Table I. For example, for the "Incumbent" dataset, employees are usually not assigned to new projects on random days, the assignments tend to happen at the beginning of a week or month. For the "Feed" dataset, multiple measurements (which are valid until the next one is made) are taken in the course of a day, resulting in a whole batch of intervals starting at the same time. The clustering is not just due to the relatively coarse granularity (one day) of these two datasets. The "Webkit" repository dataset, which looks at intervals in which files are not modified, has a granularity measured in milliseconds. Still we observe a clustering of starting events: a commit usually affects and modifies several files. The "Flight" dataset, which has granularity of minutes, also exhibits a similar pattern in the form of batched departure times.

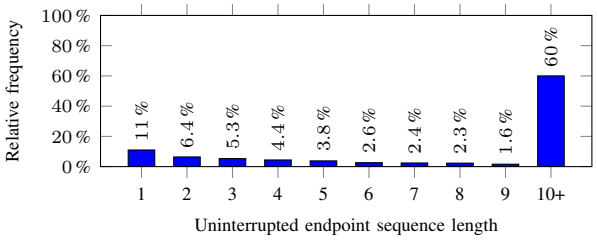


Fig. 8. Distribution of uninterrupted sequence lengths for self-join of the "inc" dataset

Reduction Factor: The real performance implication is that LEBI-Join executes fewer `getnext` operations than EBI-Join in such a scenario. The actual reduction depends not only on the clusteredness of the starting events, but also on the size of the corresponding active tuple set and the buffer capacity of LEBI-Join. We define a `getnext` *operation reduction factor* ($GNORF$), changing the cost c_z for the LEBI-Join to

$$c_z = \frac{n_z \cdot c_{getnext}}{GNORF_{rs}}.$$

For the self-join of the "Incumbent" dataset and for buffer capacity of 32 the $GNORF$ is equal to 23.6, which corresponds to huge savings in terms of run time. We also calculated this statistic for self-joins of other real-world datasets ("feed": 31.2, "web": 9.73, "flight": 7.14). Even for self-joins we get a considerable reduction factor: when encountering multiple starting events with the same timestamp, we first deal with all those of one relation and then with those of the other relation.

Join Performance: Next we compare EBI-Join with LEBI-Join, investigating the relative performance of an actual join operation (see Figure 9). LEBI-Join clearly outperforms EBI-Join, achieving a relative speedup of up to nine.

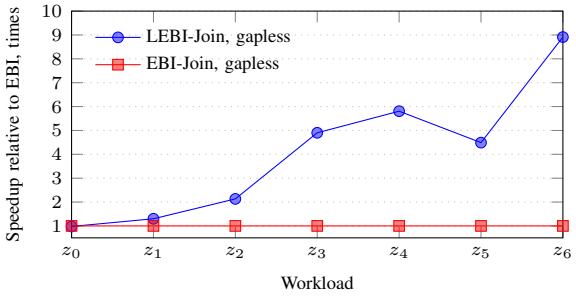


Fig. 9. EBI-Join vs LEBI-Join, synthetic data

For real-world datasets the differences can also be drastic. Figure 10 depicts the results for the "Incumbent" (inc), "Webkit" (web), "Feed" (feed), and "flight" datasets, showing that LEBI-Join outperforms EBI-Join by up to a factor of eight. Therefore, we drop EBI-Join from the diagrams from here on.

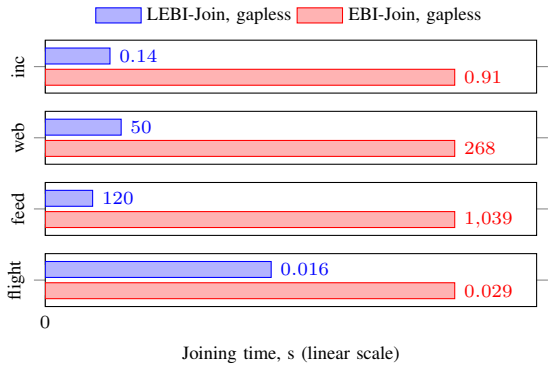


Fig. 10. EBI-Join vs LEBI-Join, real-world data

2) Comparison to the State-of-the-art: First, we present the results for the synthetic datasets (see Figure 11). For small interval durations (left-hand side of the figure) we have a small number of result tuples and preprocessing such as sorting

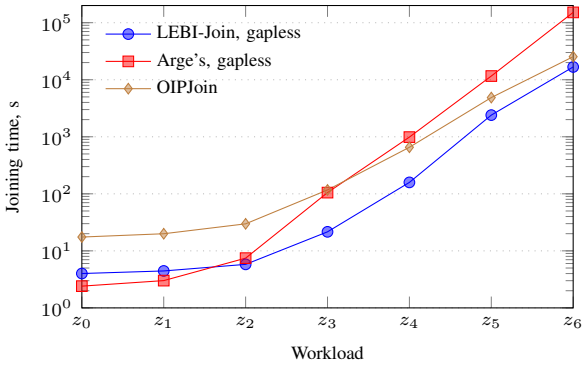


Fig. 11. Algorithm comparison, synthetic data

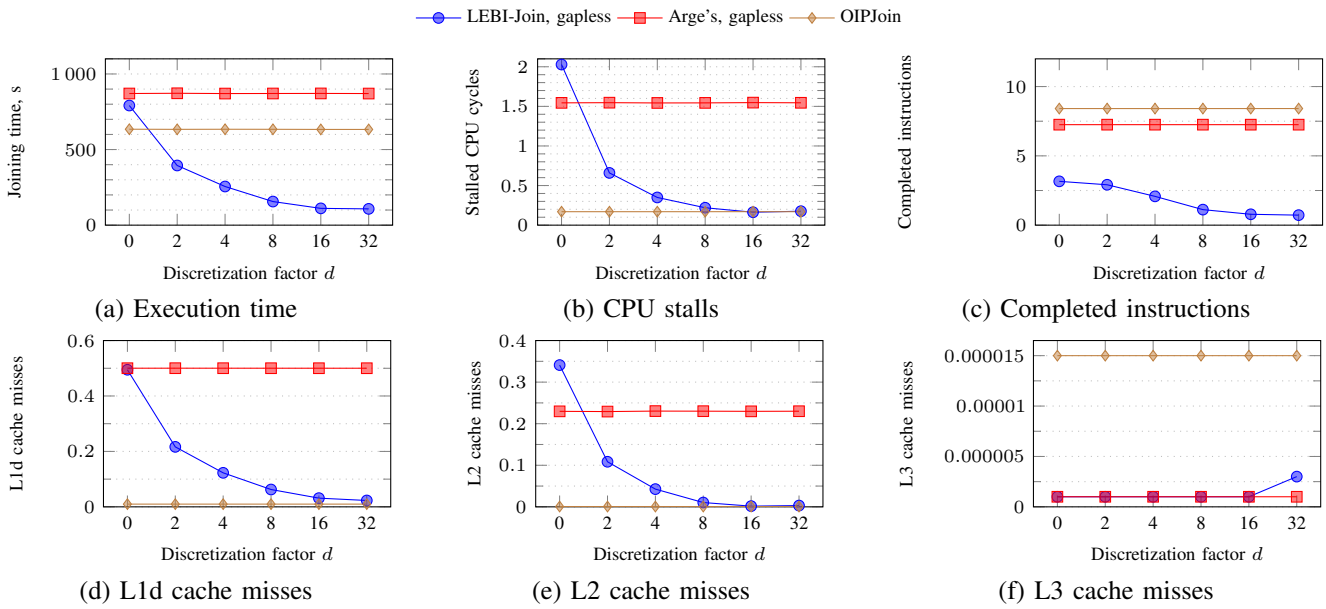
and indexing severely outweighs the time needed for the join itself. For such cases Arge's interval join performs very well, since it only needs to do a simple sort on the base relations. The performance of Arge's interval join quickly decreases for datasets with a higher number of active tuples. Even though we implemented it using a gapless list for storing the active tuples, it still has to scan the active tuples for each input tuple. LEBI-Join, on the other hand, benefits from lazy joining. OIPJoin is only competitive for very large interval durations because for short intervals it has to rule out many false positives and has a large overhead for searching and accessing a large number of small partitions.

Next, we show what happens when we vary the number of distinct points and also explain the performance of the algorithms in more detail. The first workload consists of an outer relation that contains 10^7 tuples that have exponentially distributed interval durations with an average of $5 \cdot 10^6$. The starting endpoints were uniformly distributed within the domain $[1, 10^9]$. The inner relation is a copy of the outer, but with all intervals shifted by one time point to the right. This gave us a workload with $GNORF \approx 1$. We called this workload $d = 0$. For the workload $d = 2$ we (independently) discretized the domain of each relation into $5 \cdot 10^6$ granules so that on average two tuples start (and end) at the same time point. We continued the discretization by creating coarser granules, thereby increasing the number d of tuples starting (and ending) at the same time point. Table II shows the different values for d we used, each with its corresponding reduction factor $GNORF$. The size of the output is still roughly 10^{12}.

TABLE II. DISCRETIZATION WORKLOADS

d	$GNORF$	d	$GNORF$
0	1.012	8	8.023
2	2.315	16	16.038
4	4.084	32	21.956

Figure 12(a) shows that in terms of total run time LEBI-Join is the only algorithm profiting from an increase of the factor d. However, when investigating the stalled CPU cycles per output tuple (Figure 12(b)), we are in for a surprise. While LEBI-Join can decrease the number of stalled CPU cycles for larger values of d (resulting in a better execution time), OIPJoin already has a fairly small number of idle cycles, so its run time cannot be explained by a stalled CPU. Figure 12(c) explains this phenomenon: OIPJoin is a more complex algorithm having to execute a larger number of instructions per output tuple (it has to rule out false positives). LEBI-Join not only reduces



(a) Execution time (b) CPU stalls (c) Completed instructions

(d) L1d cache misses (e) L2 cache misses (f) L3 cache misses

Fig. 12. Discretization experiments

CPU-stalls for an increasing factor of d, it can also lower the number of executed instructions, as it spends less and less time scanning active tuple sets.

We also examined the average number of cache misses per output tuple for the three algorithms (Figure 12(d), (e), and (f)) and observed an interesting effect. While Arge's algorithm and LEBI-Join have some L1d and L2 cache misses, they very rarely have to go beyond the L3 cache. OIPJoin generally has fewer cache misses, but if there is one, it has to go all the way to RAM to fetch the data. While OIPJoin processes data within one partition, all accesses are very localized. When it changes from one partition to another, though, it has to access a completely different storage area.

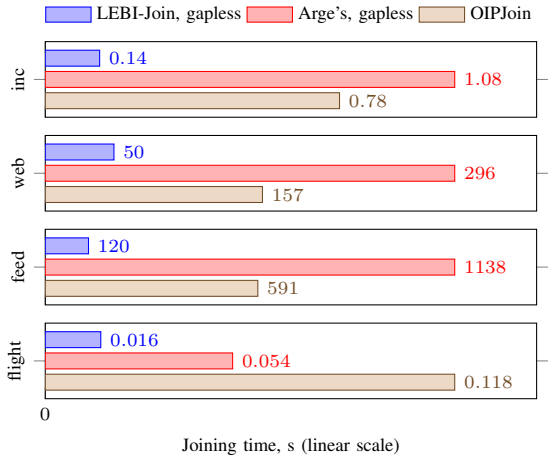


Fig. 13. Real-world dataset self-joins

Figure 13 for the real-world datasets speaks for itself. LEBI-Join is the clear winner and it beats the competition by up to an order of magnitude. High *GNORF* values of real-world data allows LEBI-Join to minimize the expensive scanning of the active tuple sets.

In the above experiments we perform self-joins. In the following experiment, we subsample one of the relations (taking every n-th tuple) to simulate a scenario in which one of the relations is smaller and has fewer active tuples (Fig. 14). Due to space constraints, we only provide the results for the "Webkit" dataset; the other datasets show a similar behavior. LEBI-Join performs well on the whole range of workloads independently of the size of the subsampled relation.

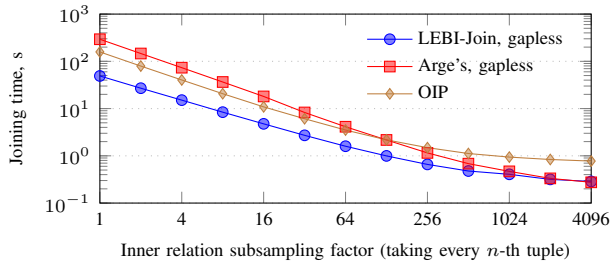


Fig. 14. Subsampling of "Webkit" dataset

3) Parallel Execution: Finally, we analyze the parallel execution of LEBI-Join (see Figure 15). We divide each relation into one, two, three and four partitions and assign every pairwise join to a task, resulting in one, four, nine, and sixteen tasks, respectively. Our benchmark machine has two CPUs with eight cores each and we can clearly see an improvement in run time for parallel execution (note the logarithmic scale). The average speedup for four, nine, and sixteen tasks is 2.5, 3.8, and 4.6, respectively.

IX. Conclusions and Future Work

Due to the proliferation of main memory database systems, the development of algorithms that use CPU caches efficiently has gained significantly in importance. Ideally, these algorithms should not be parameterized, meaning that they adapt to different cache sizes and environments automatically.

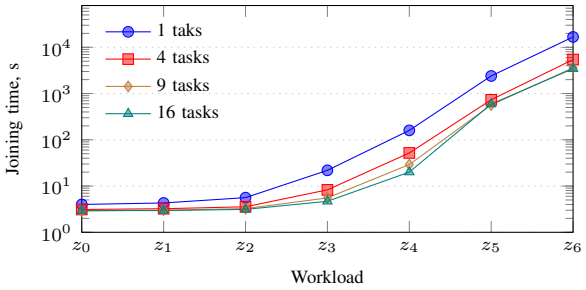


Fig. 15. Parallel execution of LEBI-Join, synthetic data

We developed a fast and robust interval join algorithm, LEBI-Join, that does not require tuning for specific hardware. At the heart of the algorithm is a data structure, gapless hash map, that manages active tuple sets during the join processing in a cache-efficient manner. We push the optimization further by reducing the number of scans we need to execute on active tuple sets. We analyze our data structures and algorithms, show where the performance gains originate, and confirm with an experimental evaluation that LEBI-Join is able to outperform the state-of-the-art algorithms for interval joins. In particular, for real-world data the difference between LEBI-Join and its competitors can reach an order of magnitude. We also show that LEBI-Join is more universal than its competitors by performing well for a very wide range of workloads.

For future work, we would like to improve the performance of other operators processing temporal or interval data by making them more cache-efficient. For example, techniques similar to ours could be used for predicates checking whether intervals are contained within each other. A more sophisticated approach would be needed for computing temporal aggregates more efficiently. Our method may even be applicable for completely different types of data, such as XML or other semistructured documents, for which algorithms and index structures relying on intervals are used. Furthermore, our algorithm is also well-suited for stream processing, as we handle events in time order. In general, we believe that there is still a lot of optimization potential when it comes to implementing interval-based operators by analyzing their behavior and adapting them to modern hardware architectures.

Acknowledgments

We thank Andreas Behrend for providing the Flight dataset.

References

[1] K. Kulkarni and J.-E. Michels, "Temporal features in SQL:2011," *SIGMOD Rec.*, vol. 41, no. 3, pp. 34–43, Oct. 2012.

[2] B. Chawda, H. Gupta, S. Negi, T. A. Faruquie, L. V. Subramaniam, and M. K. Mohania, "Processing interval joins on map-reduce," in *EDBT*, 2014, pp. 463–474.

[3] R. Cheng, S. Singh, S. Prabhakar, R. Shah, J. S. Vitter, and Y. Xia, "Efficient join processing over uncertain data," in *CIKM*, 2006, pp. 738–747.

[4] H.-P. Kriegel, P. Kunath, M. Pfeifle, and M. Renz, "Distributed intersection join of complex interval sequences," in *DASFAA*, 2005, pp. 748–760.

[5] S. K. Cha and C. Song, "P*TIME: Highly scalable OLTP DBMS for managing update-intensive stream workload," in *VLDB*, 2004, pp. 1033–1044.

[6] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots," in *ICDE*, Hannover, Germany, 2011, pp. 195–206.

[7] A.-P. Liedes and A. Wolski, "SIREN: A memory-conserving, snapshot-consistent checkpoint algorithm for in-memory databases," in *ICDE*, Atlanta, Georgia, 2006, p. 99.

[8] P. A. Boncz, M. L. Kersten, and S. Manegold, "Breaking the memory wall in MonetDB," *CACM*, vol. 51, no. 12, pp. 77–85, 2008.

[9] J. Cieslewicz, W. Mee, and K. A. Ross, "Cache-conscious buffering for database operators with state," in *DaMoN*, 2009, pp. 43–51.

[10] B. He and Q. Luo, "Cache-oblivious databases: Limitations and opportunities," *ACM TODS*, vol. 33, no. 2, pp. 8:1–8:42, Jun. 2008.

[11] S. Manegold, P. A. Boncz, and M. L. Kersten, "Optimizing main-memory join on modern hardware," *IEEE TKDE*, vol. 14, no. 4, pp. 709–730, 2002.

[12] A. Segev and H. Gunadhi, "Event-join optimization in temporal relational databases," in *VLDB*, 1989, pp. 205–215.

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

[14] T. Y. C. Leung and R. R. Muntz, "Temporal query processing and optimization in multiprocessor database machines," in *VLDB*, 1992, pp. 383–394.

[15] M. D. Soo, R. T. Snodgrass, and C. S. Jensen, "Efficient evaluation of the valid-time natural join," in *ICDE*, 1994, pp. 282–292.

[16] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter, "Scalable sweeping-based spatial join," in *VLDB*, 1998, pp. 570–581.

[17] D. Gao, C. S. Jensen, R. T. Snodgrass, and M. D. Soo, "Join operations in temporal databases," *The VLDB Journal*, vol. 14, no. 1, pp. 2–29, Mar. 2005.

[18] D. M. d. Berg, D. M. v. Kreveld, P. D. M. Overmars, and D. O. C. Schwarzkopf, "More geometric data structures," in *Computational Geometry*. Springer Berlin Heidelberg, 2000, pp. 211–233.

[19] H. Edelsbrunner, "Dynamic rectangle intersection searching," Technical University of Graz, Austria, Institute for Information Processing Report F47, 1980.

[20] J. Enderle, M. Hampel, and T. Seidl, "Joining interval data in relational databases," in *SIGMOD*, 2004, pp. 683–694.

[21] H. Samet, J. Sankaranarayanan, and M. Auerbach, "Indexing methods for moving object databases: games and other applications," in *SIGMOD*, 2013, pp. 169–180.

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

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

[24] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu, "Multi-core, main-memory joins: Sort vs. hash revisited," *PVLDB*, vol. 7, no. 1, pp. 85–96, 2013.

[25] F. P. Preparata and M. I. Shamos, *Computational geometry: An introduction*. Springer-Verlag New York, Inc., 1985.

[26] S. Moore and J. Ralph, "User-defined events for hardware performance monitoring," in *ICCS 2011 Workshop: Tools for Program Development and Analysis in Computational Science*, Singapore, 2011.

[27] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, "Quickly generating billion-record synthetic databases," in *SIGMOD*, 1994, pp. 243–252.

[28] J. A. G. Gendrano, R. Shah, R. T. Snodgrass, and J. Yang, "University information system (UIS) dataset," TimeCenter CD-1, 1998.

[29] "The webkit open source project," `http://www.webkit.org`, 2012.

[30] A. Behrend and G. Schüller, "A case study in optimizing continuous queries using the magic update technique," in *SSDBM*, 2014.