Band Joins for Interval Data

Panagiotis Bouros
Johannes Gutenberg University
Mainz, Germany
bouros@uni-mainz.de

Konstantinos Lampropoulos
University of Ioannina, Greece
klampropoulos@cs.uoi.gr

Dimitrios Tsitsigkos
University of Ioannina, Greece
Athena Research Center, Greece
dtsitsigkos@imis.athena-innovation.gr

Nikos Mamoulis
University of Ioannina, Greece
nikos@cs.uoi.gr

Manolis Terrovitis
Athena Research Center, Greece
mter@imis.athena-innovation.gr

ABSTRACT

Interval data are found in a wide range of applications (e.g., validity intervals in temporal databases, ranges of uncertain values in probabilistic databases, etc.) We study the efficient (parallel) evaluation of *band joins* for interval data. Specifically, given two collections R and S of intervals, the objective is to find all pairs (r, s), such that $r \in R$, $s \in S$, and the *difference gap* between r and s is at most equal to a given threshold ϵ. We first show how this problem can be solved by directly applying the state-of-the-art domain-based partitioning approach for interval joins, after extending the intervals by ϵ. Then, we propose a novel partitioning strategy for the original intervals, which defines the partitions using the threshold ϵ and achieves much better performance, on most datasets, for reasonably large values of ϵ.

1 INTRODUCTION

The evaluation of joins with non-equality predicates finds many applications, especially in data domains where values are approximate by nature (e.g., temporal data). For example, one application is finding pairs of events whose time difference is not greater than a given threshold ϵ. This bounded-difference join is also called *band join* [6], since for each value v in one join input, the objective is to find the values in the other input, which are inside an $[-\epsilon, \epsilon]$ band around v.

Although the evaluation of band joins has already been studied for offline (disk-resident) [6, 10] and streaming data [1, 7], previous work focuses on joins between collections of values (not intervals). In addition, the possibilities of parallel evaluation using modern hardware are not fully explored. In this paper, we study the evaluation of band joins between two collections of intervals. Specifically, given two collections R and S of intervals and a band constraint ϵ, our objective is to find all pairs (r, s) of intervals, such that $r \in R$, $s \in S$, and the *difference gap* between r and s is at most ϵ. More precisely, for two intervals $r = [r.start, r.end]$ and $s = [s.start, s.end]$ to qualify the join, it should be $s.start \leq r.end + \epsilon$ and $r.start \leq s.end + \epsilon$.

To our knowledge, this problem has not been studied before, although it has important applications. For example, the user of a temporal database [9] may often be interested in finding pairs of intervals that qualify some overlap or distance constraints (e.g., find pairs of flights which do not have a difference gap larger than 2 hours). Band joins can also be useful for *coalescing* pairs of intervals that overlap or they are close to each other [2]. Another application is in probabilistic databases, where uncertain values are often approximated by confidence intervals [5, 11]. Finding pairs of values that differ less than a threshold ϵ can be modeled and solved as a band interval join problem. XML queries can be modeled as joins between intervals that capture the positions and ancestor-descendant relationships scope of nodes in XML trees [4]. Queries over data streams [1] also constrain the time differences between events, which could be instantaneous or with a temporal duration (i.e., intervals); hence, streaming data analytics could benefit from fast band join evaluation algorithms.

In our previous work [3], we studied the parallel evaluation of *interval overlap joins*, where the objective is to find the pairs of intervals from two collections that *overlap* (i.e., share at least one value). This is a special case of the problem that we study here for $\epsilon = 0$. We proposed a *domain-based partitioning* approach, which divides the data from the two collections into partitions and processes the partitions independently and in parallel, while avoiding duplicate results.

In this work, we extend our framework to evaluate interval band joins. An intuitive and straightforward approach in this direction is to *expand* the intervals in both collections by ϵ (e.g., by adding ϵ to endpoint $x.end$ of each interval x). The interval overlap join between the two collections of expanded intervals is equivalent to the interval band join on the original data inputs and, hence, we can directly apply the original approach of [3]. On the other hand, expanding the intervals increases data replication, which could slow down the evaluation of the join.

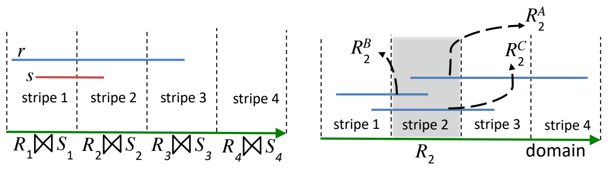
This motivated us to design an alternative approach that partitions the original intervals, as in an overlap join, but sets the width of each partition to ϵ. Our new algorithm joins each partition R_i from R with exactly two partitions from S, S_i and S_{i+1} (and vice versa), corresponding to the same and the next ϵ-wide stripes of the domain. As we show, the $R_i \bowtie S_i$ band join reduces to a cross-product, while the $R_i \bowtie S_{i+1}$ band join can be processed very efficiently, after further dividing the intervals in each partition into classes, based on the way they intersect the corresponding stripe.

We evaluate the proposed algorithm on four real datasets and varying ϵ thresholds and confirm that the ϵ-wide stripes approach is superior to the baseline adaptation of [3] on most datasets, for reasonably large values of ϵ.

2 BACKGROUND

In this section, we review the domain-based partitioning approach of [3] for interval overlap joins, which is necessary for understanding our solutions to the band interval join problem.

In order to process the join efficiently and in parallel, this approach first divides the data domain into disjoint regions (stripes),


r
s
stripe 1
stripe 2
stripe 3
stripe 4
$R_1 \bowtie S_1$ $R_2 \bowtie S_2$ $R_3 \bowtie S_3$ $R_4 \bowtie S_4$
R_2^A
R_2^B
R_2^C
R_2
domain

(a) Domain-based partitioning (b) mini-partitions

Figure 1: Example of domain-based partitioning

the union of which covers the entire domain. For each of the two input collections R and S, we then define one partition per stripe and assign each interval to all stripes that the interval overlaps. Hence, an interval may span multiple partitions. After this *partitioning phase*, partition R_i (respectively, S_i) includes all intervals from R (respectively, S) which overlap the i-th stripe. During the *join phase*, each partition R_i only has to be joined with the corresponding partition S_i. Moreover, each of the $R_i \bowtie S_i$ partition-to-partition joins can be evaluated independently from the others and the joins can be processed in parallel. Figure 1(a) illustrates an exemplary domain partitioning to four stripes and two intervals; $r \in R$ is assigned to partitions R_1, R_2, and R_3 and $s \in S$ is assigned to partitions S_1 and S_2.

However, a brute-force implementation of the *join phase* may produce *duplicate results*. For example, in Figure 1(a), join pair (r, s) would be reported by both $R_1 \bowtie S_1$ and $R_2 \bowtie S_2$. Duplicates can be avoided by reporting a pair (r, s) in a partition-to-partition join $R_i \bowtie S_i$ only if at least one of r or s starts inside the i-th stripe. Otherwise, the pair would also be detected in the join of a previous stripe. Hence, in our example, pair (r, s) is reported by $R_1 \bowtie S_1$, but the pair is *pruned*, after being detected by $R_2 \bowtie S_2$.

Instead of eliminating duplicates this way, the approach of [3] goes one step further, by avoiding the generation of such duplicates overall. The idea is to further divide each partition R_i into three *mini-partitions* R_i^A, R_i^B, and R_i^C; R_i^A takes all intervals in R_i which start in stripe i, R_i^B takes all intervals in R_i which start before stripe i and end inside stripe i, and R_i^C takes all intervals that start before stipe i and end after stripe i. Figure 1(b) shows examples of three intervals from R_2 that go to different mini-partitions. Now, each $R_i \bowtie S_i$ can be computed by performing 5 *mini-joins* between the mini-partitions, as shown in Figure 2:

- $R_i^A \bowtie S_i^A$ is computed as a typical interval overlap join;
- $R_i^A \bowtie S_i^B$ and $R_i^B \bowtie S_i^A$ are computed as a special case of an interval join, where the start point of every interval in S_i^B (resp. R_i^B) precedes all start points of all intervals in R_i^A (resp. S_i^A) [3].
- $R_i^A \bowtie S_i^C$ and $R_i^C \bowtie S_i^A$ are cross products, hence their computation requires no comparisons;
- $R_i^B \bowtie S_i^B$, $R_i^B \bowtie S_i^C$, $R_i^C \bowtie S_i^B$, and $R_i^C \bowtie S_i^C$ *do not have to be computed* because they would produce duplicate join results (guaranteed to be found in previous stripes).

Mini-joins are evaluated by an optimized version of a *forward scan* algorithm based on plane-sweep (also proposed in [3]).

3 EVALUATING BAND JOINS

3.1 Evaluation based on interval overlap joins

As discussed in the Introduction, a baseline evaluation algorithm for interval *band* joins transforms the problem to an interval *overlap* join. For this purpose, it expands the intervals from both input collections by ϵ. Without loss of generality, each interval $r \in R$ and $s \in R$ becomes $r' = [r.start, r.end + \epsilon]$ and $s' = [s.start, s.end + \epsilon]$, respectively. We can easily show that if r' overlaps s' then $s.start \leq r.end + \epsilon$ and $r.start \leq s.end + \epsilon$ hold, i.e., pair (r, s) satisfies the band join predicate.

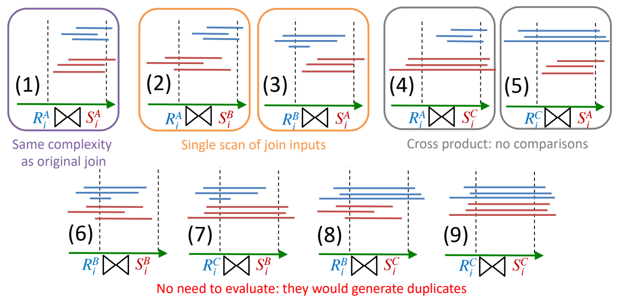


Figure 2: Breakdown of $R_i \bowtie S_i$ into mini-joins

This baseline can be straightforwardly implemented using the approach of [3]. The expansion of the input intervals takes place before they are assigned to the partitions, while the mini-joins breakdown operates exactly as discussed in the previous section.

3.2 Evaluation on ϵ-wide partitions

Despite its simplicity, the baseline exhibits two shortcomings. First, due to expanding intervals by ϵ,data replication increases (compared to the replication in the overlap interval join problem). This increases the cost of the partition-to-partition joins. The second drawback is that the domain-based partitioning is agnostic to the input parameter ϵ. For instance, a pair of intervals located at the two different ends of a stripe may not qualify the band join predicate; nevertheless, they need to be checked.

To address these issues, we next propose our second solution for band joins. The key idea of the method is to split the domain into disjoint ranges (stripes), such that the width of each stripe is ϵ (in case the domain cannot be divided exactly by ϵ, the last stripe is narrower). The input intervals from R (resp. S) are not expanded, but directly assigned to every partition R_i (resp. S_i) they intersect, as described in Section 2.

Since the width of each stripe i is (at most) ϵ, it is guaranteed that every pair of interval (r, s) with $r \in R_i, s \in S_i$ forms a result of the band join. In other words, $r.end + \epsilon < s.start$ or $s.end + \epsilon < r.start$ cannot hold; otherwise, r and s would not have been assigned to the same partition. Hence, we can directly report all pairs (r, s) with $r \in R_i, s \in S_i$ as results. However, the same pair of intervals could co-exist in other stripes as well (e.g., the $(i-1)$-th and/or the $(i+1)$-th). Therefore, as in the interval overlap join case, we should only report a pair if it is not a join result in a previous stripe, i.e., if at least one of r or s start inside stripe i. To avoid this test, we can divide each partition R_i (and S_i) again into three mini-partitions R_i^A, R_i^B, R_i^C (and S_i^A, S_i^B, S_i^C), as explained in Section 2 and then evaluate *all* $R_i^A \bowtie S_i^A$, $R_i^A \bowtie S_i^B$, $R_i^B \bowtie S_i^A$, $R_i^A \bowtie S_i^C$, and $R_i^C \bowtie S_i^A$ mini-joins as *cross-products*.

However, we are not done yet. There could also be band join results (r, s), such that r *ends* in stripe i, s *starts* in stripe $i + 1$ and $r.end + \epsilon \leq s.start$ (and the symmetric case). The current decomposition has no explicit partition for all intervals that end in stripe i, i.e., the mini-partition R_i^A does not distinguish between the intervals $r \in R$ that end in stripe i from those that end after stripe i. To this end, we define an *additional* mini-partition R_i^{A1}, which contains the intervals $r \in R_i$ that both *start* and *end* inside stripe i. Mini-partition R_i^{A1} is a subset of R_i^A, but their contents are sorted differently to enhance the join evaluation, as we discuss in the next paragraph. Figure 3 shows examples of four intervals

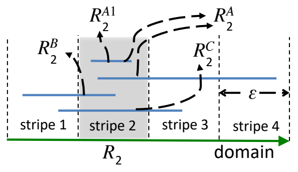
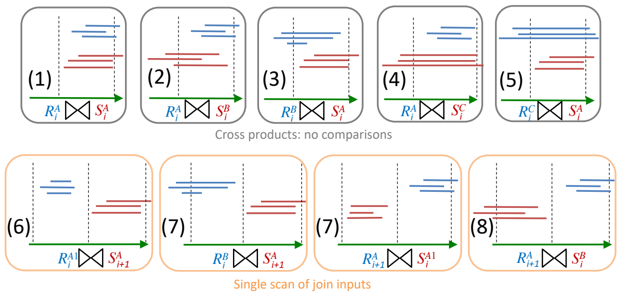


Figure 3: Mini-partitions for band joins



(1) $R_i^A \bowtie S_i^A$
(2) $R_i^A \bowtie S_i^B$
(3) $R_i^B \bowtie S_i^A$
(4) $R_i^A \bowtie S_i^C$
(5) $R_i^C \bowtie S_i^A$
Cross products: no comparisons
(6) $R_i^{A1} \bowtie S_{i+1}^A$
(7) $R_i^B \bowtie S_{i+1}^A$
(7) $R_{i+1}^A \bowtie S_i^{A1}$
(8) $R_{i+1}^A \bowtie S_i^B$
Single scan of join inputs

Figure 4: Mini-joins breakdown in band joins

which are assigned to R_2^A, R_2^B, R_2^C, and R_2^{A1}. Observe that intervals which are assigned to R_2^{A1} are also assigned to R_2^A.

In Figure 4, we illustrate the set of mini-joins that have to be evaluated by the new algorithm. Besides the 5 cross-products between mini-partitions in the same stripe, we have to perform four mini-joins $R_i^{A1} \bowtie S_{i+1}^A$, $R_i^B \bowtie S_{i+1}^A$, $S_i^{A1} \bowtie R_{i+1}^A$, $S_i^B \bowtie R_{i+1}^A$, at every pair i and $i+1$ of neighboring partitions. All these mini-joins can be evaluated efficiently in a similar manner as mini-join $R_i^B \bowtie S_i^A$ of the interval overlap join problem (illustrated in Figure 2). Specifically, the intervals in mini-partitions R_i^{A1}, R_i^B, S_i^{A1}, S_i^B are sorted by $r.end$ or $s.end$ and the intervals in mini-partitions R_i^A and S_i^A are sorted by $r.start$ and $s.start$, respectively. Then, at each of the four mini-joins, we can compute the result by *concurrently scanning* the join inputs only once.

Consider, for instance, the mini-join between R_i^{A1} (sorted by $r.end$) and S_{i+1}^A (sorted by $r.start$). We first check whether $r.end + \epsilon \geq s.start$ holds for the first r in R_i^{A1} and the first s in S_{i+1}^A. If so, (r, s) is a band join result. At the same time, we can conclude that for all intervals r' that follow r in R_i^{A1}, (r', s) is also a band join result, since $r'.end \geq r.end$ holds (due to sorting). Note that all these join results are generated without any comparisons. Afterwards, we advance to the next interval $s \in S_i^A$ and repeat the test $r.end + \epsilon \geq s.start$. If the test is negative, we advance to the next interval $r \in R_i^{A1}$ and repeat the test, etc. As soon as either r or s is out of bounds, i.e., the input mini-partitions are fully scanned, the join algorithm terminates.

For example, in Figure 5, we first consider intervals r_1 and s_1. As $r_1.end + \epsilon \geq s_1.start$ holds, all intervals in R_i^{A1} form band join pairs with s_1. Next, we examine s_2 and repeat the test to find again that $r_1.end + \epsilon \geq s_2.start$. Hence, we also report all intervals in R_i^{A1} paired with s_2 as band join results. However, when we advance to s_3, we observe that $r_1.end + \epsilon < s_3.start$, so (r_1, s_3) is not a join result. At this point, we consider the next intervals r from R_i^{A1} while $r.end+\epsilon < s.start$ holds and stop at r_3, where we have $r_3.end + \epsilon \geq s_3.start$. Again, we report join pairs (r_3, s_3) and (r_4, s_3) and advance to s_4. Since $r_3.end + \epsilon < s_4.start$, we finally advance to r_4; since, $r_4.end + \epsilon \geq s_4.start$ holds, we report join pair (r_4, s_4). At this point, both mini-partitions are completely scanned and the algorithm terminates.

The number of comparisons conducted by the above algorithm (applied for all mini-joins which are not cross products) equals

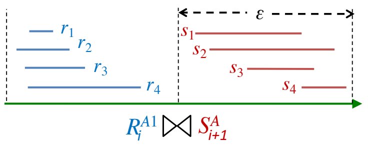


Figure 5: Mini-join between neighboring partitions

Table 1: Statistics of datasets

	INFECTIOUS	BOOKS	TAXIS	WEBKIT
Cardinality	415, 912	2, 312, 602	14, 212, 261	2, 347, 346
Domain duration (secs)	6, 946, 360	31, 507, 200	2, 592, 000	461, 829, 284
Distinct domain points	81, 514	5, 330	2, 229, 932	174, 471
Shortest interval (secs)	20	1	1	1
Avg. interval dur. (secs)	20	2, 201, 320	685	33, 206, 300
Longest interval (secs)	20	31, 406, 400	1, 816, 164	461, 815, 512

the total number of intervals in its two inputs (e.g., $|R_i^{A1}| + |S_{i+1}^A|$), which means that mini-joins are evaluated very efficiently.

Parallel evaluation. As shown in [3], the best approach to parallelize interval joins based on domain-based partitioning is to treat every mini-join as an independent task. Each task is scheduled to one of the available CPU threads. To maximize load balancing, the tasks are greedily assigned to threads in decreasing order of their expected costs (based on the size of the involved partitions).

4 EXPERIMENTAL EVALUATION

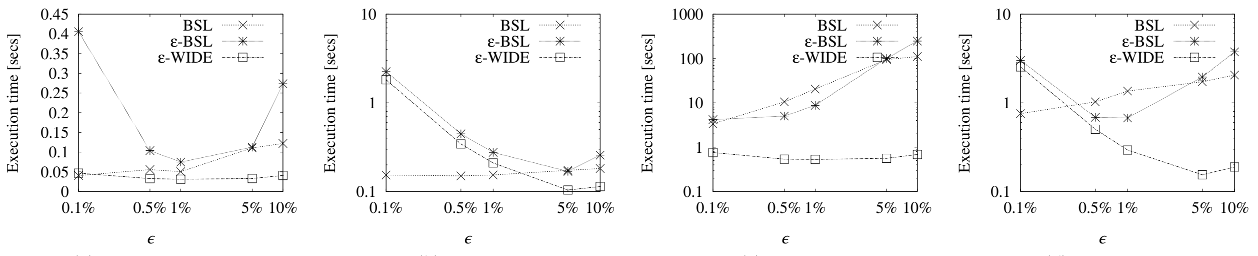
Our evaluation was conducted on a machine with 384 GBs of RAM and a dual Intel(R) Xeon(R) CPU E5-2630 v4 clocked at 2.20GHz. All methods were implemented in C++, compiled using `gcc` (v4.8.5) with flags `-O3`, `-mavx` and `-march=native`. We activated hyper-threading, allowing us to run up to 40 threads and used OpenMP for multi-threaded processing. Every interval contains two 64-bit domain point attributes (i.e., `start` and end) while the workload accumulates the number of result pairs. All data reside in main memory.

Methods. We compare our ϵ-wide partitioning join (denoted by ϵ-WIDE) to the baseline (denoted by BSL). In addition, we include a version of BSL (denoted by ϵ-BSL), which uses ϵ-wide domain partitions, but it conducts an overlap join using the extended intervals, instead of the method described in Section 3.2. Both variants of the baseline use our optimized forward scan based plane sweep method from [3] and all our optimizations to improve load balancing in domain-based partitioning.

Datasets. Table 1 details our 4 real-world experimental datasets. INFECTIOUS [8] stores contact intervals between visitors at an exhibition at the Science Gallery in Dublin from 2009/05 to 2009/07. BOOKS [3] includes periods of book lent outs at Aarhus public libraries in 2013 (https://www.odaa.dk). TAXIS (https://www1.nyc.gov/site/tlc/index.page) stores durations of taxi trips in NYC, in Jan 2013. WEBKIT [3] records durations of file versions in the git repository of the Webkit project from 2001 to 2016 (https://webkit.org).

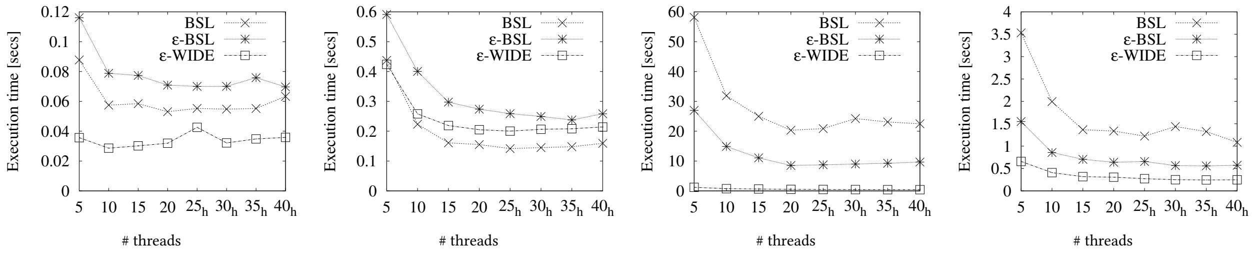
Tests. To assess the performance of the methods, we measure their response time while varying (i) threshold ϵ as a fraction of the domain duration inside $\{0.001, 0.005, 0.01, 0.05, 0.1\}$ and (ii) the number of available CPU threads inside $\{5, 10, 15, 20, 25, 30, 35, 40\}$. We also experimented with uniformly sampled subsets of the dataset as R and set the entire dataset as S; for this purpose, we varied the $|R|/|S|$ ratio inside $\{0.25, 0.5, 0.75, 1\}$.

Results. Figures 6-8 summarize our experimental results. When varying ϵ, we observe the following. First, ϵ-WIDE is consistently faster than ϵ-BSL. This shows that applying the mini-joins



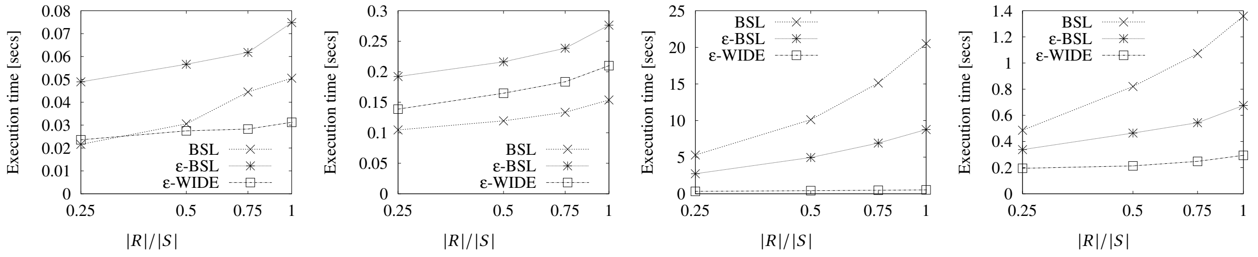
(a) INFECTIOUS (b) BOOKS (c) TAXIS (d) WEBKIT

Figure 6: Total execution time while varying threshold ϵ as fraction of the domain duration; 20 threads, $|R| = |S|$.



(a) INFECTIOUS (b) BOOKS (c) TAXIS (d) WEBKIT

Figure 7: Total execution time while varying the number of CPU threads; $\epsilon = 0.01$ of the domain size, $|R| = |S|$.



(a) INFECTIOUS (b) BOOKS (c) TAXIS (d) WEBKIT

Figure 8: Total execution time while varying the $|R|/|S|$ ratio; $\epsilon = 0.01$ of the domain size, 20 threads.

described in Section 3.2 on the original intervals is faster than applying the mini-joins of [3] on the ϵ-extended intervals. However, the number of ϵ-wide partitions can become extremely large for small values of ϵ, which renders these approaches slower compared to BSL. Note that BSL chooses the number of partitions, according to the number of threads that can run in parallel as suggested in [3]. On WEBKIT, TAXIS and INFECTIOUS, ϵ-WIDE is up to a few time faster compared to BSL and ϵ-BSL for a wide range of ϵ/domain ratios. On the other hand, on dataset BOOKS, ϵ-WIDE is faster than the competition for ϵ/domain ratios larger than 2%. When varying the number of threads, we observe that all methods scale well until when the number of threads becomes 20, after which hyper threading comes into effect. The join on INFECTIOUS is already very cheap and does not benefit from increasing parallelism. Last, as expected all methods are affected by increasing $|R|/|S|$; their execution time rises.

5 CONCLUSION

In this short paper, we studied the evaluation of band joins between two collections of intervals. We extended our framework for interval overlap joins [3] in two directions; a baseline approach that expands all intervals by ϵ and then evaluates an overlap join and a novel approach that uses ϵ to define the partitions and then conducts cheaper joins between partitions. Our experimental findings show that the second approach is more efficient, unless ϵ is very small compared to the domain size.

ACKNOWLEDGEMENTS

Partially funded by the European Union's Horizon 2020 research - innovation programme under grant agreement No 657347 and the European Regional Development Fund - GSRT's project "Moving from Big Data Management to Data Science" (MIS 5002437/3) and Research–Create–Innovate project "Proximiot" (T1EDK-04810).

REFERENCES

[1] Pankaj K. Agarwal, Junyi Xie, Jun Yang, and Hai Yu. 2005. Monitoring Continuous Band-Join Queries over Dynamic Data. In *ISAAC*. 349–359.
[2] Michael H. Böhlen, Richard T. Snodgrass, and Michael D. Soo. 1996. Coalescing in Temporal Databases. In *VLDB*. 180–191.
[3] Panagiotis Bouros and Nikos Mamoulis. 2017. A Forward Scan based Plane Sweep Algorithm for Parallel Interval Joins. *PVLDB* 10, 11 (2017), 1346–1357.
[4] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. 2002. Holistic twig joins: optimal XML pattern matching. In *SIGMOD*. 310–321.
[5] Reynold Cheng, Ben Kao, Sunil Prabhakar, Alan Kwan, and Yi-Cheng Tu. 2005. Adaptive Stream Filters for Entity-based Queries with Non-Value Tolerance. In *VLDB*. 37–48.
[6] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. 1991. An Evaluation of Non-Equijoin Algorithms. In *VLDB*. 443–452.
[7] Mohammed Elseidy, Abdallah Elguindy, Aleksandar Vitorovic, and Christoph Koch. 2014. Scalable and Adaptive Online Joins. *PVLDB* 7, 6 (2014), 441–452.
[8] Lorenzo Isella, Juliette Stehlé, Alain Barrat, Ciro Cattuto, Jean-François Pinton, and Wouter Van den Broeck. 2011. What's in a crowd? Analysis of face-to-face behavioral networks. *Journal of Theoretical Biology* 271, 1 (2011), 166–180.
[9] Christian S. Jensen and Richard T. Snodgrass. 2018. Temporal Data Models. In *Encyclopedia of Database Systems, Second Edition*.
[10] Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Panos Kalnis. 2015. Lightning Fast and Space Efficient Inequality Joins. *PVLDB* 8, 13 (2015), 2074–2085.
[11] Weining Zhang and Ke Wang. 2000. An Efficient Evaluation of a Fuzzy Equi-Join Using Fuzzy Equality Indicators. *TKDE* 12, 2 (2000), 225–237.