Short Paper

Interval Count Semi-Joins

Panagiotis Bouros
Institute of Computer Science
Johannes Gutenberg University Mainz, Germany
bouros@uni-mainz.de

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

ABSTRACT

Interval joins find applications in several domains, including temporal and spatial databases, uncertain data management, streaming data processing. In this paper, we study the evaluation of an interval count semi-join (*ICSJ*) operation that can be used for selecting or ranking intervals based on the number of join pairs they appear in. We extend the state-of-the-art algorithm for interval joins to evaluate *ICSJ* at the cost of only scanning the sorted interval endpoints.

1 INTRODUCTION

The interval join (*IJ*) is an important and well-studied operation that finds several applications. In its most widely used definition, the interval join takes as input two collections of intervals R and S, and outputs the pairs $(r, s) \in R \times S$, such that intervals r and s overlap[1]. In temporal databases [8], tuples are associated with validity intervals and interval joins can be used to find pairs of tuples with overlapping validity (e.g., find pairs of employees who worked in two enterprises during overlapping time periods). In spatial databases, multidimensional overlap joins reduce to interval joins if the spatial extent of the objects is represented by a set of intervals with the help of space-filling curves [11]. In probabilistic databases, values in continuous domains are often represented by intervals of values which have non-zero probability [4]. Finally, in applications that process streaming data, values read from different streams can be joined by (often parameterized) temporal windows [7]. Such sliding window joins can be modeled as overlap joins, if the values are extended by the window lengths and modeled as intervals. A number of single-processor [2, 5, 6, 13, 14] and parallel [1, 3, 9] algorithms for interval joins have been proposed. Among them, methods that are based on plane-sweep prevail due to their optimal worst-case complexity and their efficient implementations [1, 13].

In this paper, we study the efficient evaluation of an *Interval Count Semi-Join* (*ICSJ*) operation, where the objective is to find how many pairs in the interval join $IJ(R, S)$ result include each $r \in R$. For example, consider interval collections $R = \{r_1, r_2, r_3\}$ and $S = \{s_1, s_2, s_3, s_4\}$ depicted in Figure 1. $ICSJ(R, S) = \{(r_1, 2), (r_2, 1), (r_3, 3)\}$ because r_1, r_2 and r_3 overlap with 2, 1, and 3 intervals from S, respectively. $ICSJ(R, S)$ can be seen as a case of temporal aggregation on S, using R as the set of fixed intervals [12]. The result of *ICSJ* can be used to select or rank objects that are associated with the intervals in R based on the number of intervals in S they intersect. For example, if R includes the employment periods of employees in company A and S includes the periods of employees in company B, we may wish to find the k employees in A whose employment time overlaps with that of the most employees in B (e.g., employee r_3 in Figure 1, if $k = 1$). This problem can be solved by ranking the *ICSJ* output, or by using a priority queue to keep track of the top-k *ICSJ* results while they are computed.

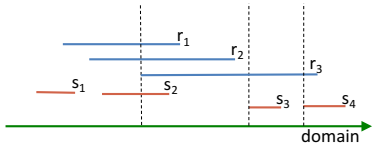

Figure 1: Collections $R = \{r_1, r_2, r_3\}$ and $S = \{s_1, s_2, s_3, s_4\}$.

To our knowledge, *ICSJ* has not been adequately studied to-date. Top-k count semi-joins have been studied in relational [15] and spatial databases [16], but the techniques in these studies are unsuited for *ICSJ*, as they apply on different data domains and they make use of indices. We present an efficient *smart counting* algorithm for evaluating *ICSJ*, which applies on two sorted input collections R and S and extends the state-of-the-art sweeping based *IJ* algorithm. The algorithm bears only the minimal cost of scanning the sorted inputs. Experiments on four real datasets show that it is orders of magnitude faster than simpler alternatives.

2 BACKGROUND

Related work on *IJ* includes techniques based on indexing or partitioning [2, 5, 6] and methods that sort inputs R, S to perform merge-join [14] or plane-sweep based join [1, 13]. Recent studies [1, 13] focused on in-memory processing and showed that plane-sweep based techniques are superior to other methods.

Algorithm 1 describes a plane-sweep based algorithm for interval joins. Initially, the domain points (or simply points) of all intervals in R, S are extracted and sorted into list L (Lines 2–3). Intuitively, L defines the stops of an imaginary line that sweeps the domain; so, the sweep line stops both at the start and the end point of an interval. An *active set* A^R and A^S is initialized for each of the two input collections. Sets A^R and A^S keep track of the intervals that are currently "open" (i.e., their start point has been encountered but not their end point). Each point in list L is accessed in order; if it is a start point of an interval, e.g., $r \in R$, r is guaranteed to overlap all intervals in A^S. Therefore, all pairs in $\{r\} \times A^S$ are reported. For example, consider Figure 1 and assume that the start point of r_3 is currently accessed (i.e., the sweep line is the leftmost vertical line). The active set of S is $A^S = \{s_2\}$, hence the algorithm outputs pair (r_3, s_2) as part of the join result. If an end point is encountered by the sweep line (Lines 11–12 and 18–19 of Algorithm 1), the corresponding interval is no longer open, so it is removed from its respective active set.

Assuming an efficient data structure, which performs insertions and deletions to the active sets in constant time (e.g., a hash table), Algorithm 1 computes the join in $O(|R| + |S| + K)$ time, where K is the number of result pairs, excluding the sorting cost of list L. Note that when a start point is encountered, an active set should be scanned to generate join results. Scanning a hash table

[1] Two intervals overlap (or intersect) if they share at least one common value.

ALGORITHM 1: Plane-Sweep based Interval Join

Input : collections of intervals R and S
Output : set of all intersecting interval pairs $(r, s) \in R \times S$
Variables : interval points list L, active sets A^R, A^S

```
1  A^R ← ∅, A^S ← ∅; ▷ sets of active intervals from R and S
2  L ← start and end points of all intervals in R ∪ S;
3  sort L;
4  while L is not depleted do
5      p ← next point in L;
6      if p originates from collection R then
7          r ← interval in R where p belongs;
8          if p is a start point then
9              add r to A^R;                          ▷ r is open
10             output {(r, s) : ∀s ∈ A^S};            ▷ r overlaps all
                   intervals in A^S
11         else remove r from A^R;                    ▷ r no longer open
12     else
13         s ← interval in S where p belongs;
14         if p is a start point then
15             add s to A^S;                          ▷ s is open
16             output {(r, s) : ∀r ∈ A^R};            ▷ s overlaps all
                   intervals in A^R
17         else remove s from A^S;                    ▷ s no longer open
```

is expensive as it incurs random accesses in memory; to this end, Piatov et al. [13] designed the *gapless hash map* which efficiently supports all three `insert`, `remove` and `getNext` operations.[2] Finally, in [1], an implementation of the plane-sweep based interval join that replaces active sets (e.g., A^R) by forward scans to the other collection (i.e., S), was investigated and optimized.

3 EVALUATING *ICSJ*

We now investigate how the plane-sweep based Algorithm 1 can be extended to efficiently evaluate interval count semi-joins. Recall that the goal is to count for every interval $r \in R$, the number of overlapping intervals from S.

A *naïve* approach for computing $ICSJ(R, S)$ is to evaluate $IJ(R, S)$ first. Then in an aggregation step, we need to sort or hash the (r, s) join pairs by their first element, and count and report the number of pairs for each $r \in R$. Naturally, this method is at least as expensive as the interval join problem. In fact, since the number of overlapping intervals can be much greater than the sizes $|R|$ and $|S|$ of the two inputs, the cost of sorting or hashing the $IJ(R, S)$ results may dominate the overall evaluation cost.[3]

To address these shortcomings, we next present two methods, which extend Algorithm 1 to directly compute $ICSJ(R, S)$.

3.1 The Simple Counting Approach

We first discuss an intuitive extension to Algorithm 1 based on the following observation. When the start point of an interval (e.g., $r \in R$) is encountered, the algorithm scans the active set of the other collection (i.e., A^S) to produce join pairs $\{r\} \times A^S$. As our objective is only to count the intervals from S that overlap interval r, we do not have to scan active set A^S; instead, we only need to add its size $|A^S|$ to a dedicated counter for r. Note that

[2] In [13], Algorithm 1 is presented as the *Endpoint-Based Interval* (EBI) Join algorithm.
[3] Note also that this naïve method is not suitable for in-memory evaluation of count semi-joins due to buffering all $IJ(R, S)$ result pairs.

ALGORITHM 2: Simple Counting

Input : collections of intervals R and S
Output : for each $r \in R$, $|s : s \in S$, and r intersects $s|$
Variables : interval points list L, active sets A^R, A^S, hash table C

```
1  A^R ← ∅, A^S ← ∅; ▷ sets of active intervals from R and S
2  L ← start and end points of all intervals in R ∪ S;
3  sort L;
4  while L is not depleted do
5      p ← next point in L;
6      if p originates from collection R then
7          r ← interval in R where p belongs;
8          if p is a start point then
9              add r to A^R;                          ▷ r is open
10             C[r] ← |A^S|;                          ▷ initialize counter for r
11         else
12             remove r from A^R;                     ▷ r no longer open
13             output (r, C[r]);
14             delete C[r];
15     else
16         s ← interval in S where p belongs;
17         if p is a start point then
18             add s to A^S;                          ▷ s is open
19             foreach r ∈ A^R do
20                 C[r] ← C[r] + 1;                   ▷ s overlaps all
                       intervals in A^R
21         else remove s from A^S;                    ▷ s no longer open
```

this is not the final value of this counter, because r may also overlap with intervals from S that start later. Nevertheless, we can eliminate the overhead of scanning the active set of collection S, which in practice also means that a typical hash table can be used for A^S instead of an optimized special structure (e.g., the gapless hash map of [13]) as we only need to support efficient insertions and deletions. In contrast, we still have to support efficient scans for active set A^R, because for each encountered start point from S, we have to scan A^R in order to increase the counters of all open intervals from r.

Algorithm 2 is a pseudocode of this *Simple Counting* approach. Compared to Algorithm 1, we define a hash table C to maintain the dedicated counter for each open interval from collection R. Further, as already discussed, Simple Counting initializes counter $C[r]$ in Line 10, when the start point of an interval $r \in R$ is seen. The counter for each $r \in A^R$ is then increased by 1, when the start of an interval from S is seen in Lines 19–20. Finally, as soon as the end point of r is accessed, counter $C[r]$ is finalized and hence removed from hash table C and reported as result (Lines 13–14). Consider for example r_3 in Figure 1; when accessing the start point of the interval, counter $C[r_3]$ is initialized to 1 as $A^S = \{s_2\}$. After the next two stops of the sweep line marked in the figure, i.e., when the start points of intervals s_2 and s_3 are encountered, $C[r_3]$ is increased to 3. Algorithm 2 is similar to the general approach for temporal aggregation, proposed in [12].

Simple Counting is expected to always outperform the naïve solution; recall that the latter needs to completely evaluate IJ as its first step. On the other hand, Algorithm's 2 cost is in same order to Algorithm 1 as half of the IJ results are still computed, i.e., the pairs generated in Lines 19–20 when encountering start points from S. Note that Simple Counting is also charged with the book-keeping cost for the counters of hash table C. In view

ALGORITHM 3: Smart Counting

Input : collections of intervals R and S
Output : for each $r \in R$, $|s : s \in S$, and r intersects $s|$
Variables : interval points list L, hash table C

```
1  |A^S| ← 0;        ▷ active set counter for intervals from S
2  g ← 0;                                        ▷ global counter
3  L ← start and end points of all intervals in R ∪ S;
4  sort L;
5  while L is not depleted do
6      p ← next point in L;
7      if p originates from collection R then
8          r ← interval in R where p belongs;
9          if p is a start point then
10             C[r] ← |A^S| − g;    ▷ initialize counter for r
11         else
12             C[r] ← C[r] + g;
13             output (r, C[r]);
14             delete C[r];
15     else
16         s ← interval in S where p belongs;
17         if p is a start point then
18             |A^S| ← |A^S| + 1; ▷ increase active set counter
19             g ← g + 1;           ▷ increase global counter
20         else
21             |A^S| ← |A^S| − 1; ▷ decrease active set counter
```

of these shortcomings, we next present a significantly faster extension to Algorithm 1.

3.2 The Smart Counting Approach

The main idea behind the *Smart Counting* extension to Algorithm 1 is to maintain cheap statistics about the intervals from S instead of keeping track of A^S at every position of the sweep line. Algorithm 3 is the pseudocode of the Smart Counting approach. We now discuss its key features.

First, we observe that only the size of active set A^S is in fact needed for the *ICSJ* computation. Although the Simple Counting algorithm presented in Section 3.1 keeps track of the open intervals from S, the contents of A^S are never scanned and only $|A^S|$ is used on Line 10 of Algorithm 2. Hence, we can replace the hash table of active set A^S by a simple size counter $|A^S|$; when the start point of an interval $s \in S$ is encountered, this counter is increased by 1 (Line 18) while after an end point from S is accessed the same counter is reduced by 1 (Line 21). Next, we define a *global counter* g to keep track of the *number* of intervals from S that have opened (regardless whether their end point is already accessed or not). Similar to $|A^S|$, counter g is increased by 1 in Line 19 when a start point from collection S is seen but never decreased which means that $g \geq |A^S|$ always holds.

By combining counters $|A^S|$ and g, we are able to compute the number of intervals from S that opened or were open in-between the start and the end point of an interval $r \in R$. In specific, we initialize the dedicated counter $C[r] = |A^S|$ when the start point of r is encountered but then subtract the value of global counter g (Line 8). Compared to Algorithm 2, notice that we no longer maintain open intervals from R to active set A^R; instead we employ hash table C to store the current value of r's dedicated counter. After the end point of interval r is seen, we just need to add back the current value of g to $C[r]$ and report result $(r, C[r])$ (Lines 12–13). This procedure guarantees that we will end up with the correct value of $C[r]$, because the difference from global counter g corresponds to the number of intervals from S that opened after r's start point. Note that these intervals overlap with r but were not considered when $C[r]$ was initialized.

We expect the Smart Counting approach to significantly outperform Simple Counting as the cost of maintaining and scanning active sets A^R, A^S is completely eliminated. Further, we manage to avoid the random accesses that incur during the for-loop on Lines 19–20 of Algorithm 2 when the counters for multiple intervals are concurrently updated. Overall, the cost of Smart Counting (excluding sorting) is $O(|R| + |S|)$ due to the constant-time cost of processing at each position of the sweep line.

Table 1: Characteristics of experimental datasets

	FLIGHTS	BOOKS	GREEND	WEBKIT
Cardinality	445,827	2,312,602	110,115,441	2,347,346
Domain duration (secs)	2,750,280	31,507,200	283,356,410	461,829,284
Shortest interval (secs)	1,261	1	1	1
Longest interval (secs)	42,301	31,406,400	59,468,008	461,815,512
Avg. interval duration (secs)	8,791	2,201,320	16	33,206,300
Distinct domain points	41,975	5,330	182,028,123	174,471

4 EXPERIMENTAL ANALYSIS

4.1 Setup

For our experiments, we implemented all methods in C++ and compiled them using gcc (v5.2.1). Note that all data (input collections, active sets, interval points list etc.) resided in main memory.

Methods. Besides gapless hash map, the authors in [13] also discussed a lazy optimization for plane-sweep based Algorithm 1, which buffers consecutive start points in list L from the same input (e.g., R). When producing IJ results, a single scan over the active set of the other input (i.e., A^S) is performed for the entire buffer. By restricting buffers to fit inside L1 cache or even the cache registers, this technique reduces cache misses. To enhance *ICSJ* computation, we applied this lazy optimization on Naïve and Simple Counting. For the latter, we buffer consecutive start points from S allowing us to increase $C[r]$ for each $r \in A^R$ by the buffer size instead of 1 as in Lines 19–20 of Algorithm 2. On the other hand, lazy optimization has no effect on Smart Counting.

Datasets. Table 1 details our 4 real-world experimental datasets. FLIGHTS records domestic flights in USA during January 2016 (https://www.bts.gov); valid times indicate the duration of a flight. BOOKS records the transactions at Aarhus public libraries in 2013 (https://www.odaa.dk); valid times indicate the periods when a book is lent out. GREEND [10, 13] records power usage information in households across Austria and Italy from January 2010 to October 2014; valid times indicate the period of a measurement. WEBKIT records the file history in the git repository of the Webkit project from 2001 to 2016 (https://webkit.org); valid times indicate the periods when a file did not change.

Tests. We ran interval count semi-joins using a uniformly sampled subset of each dataset as outer input R and the entire dataset as inner S; for this purpose, we varied ratio $|R|/|S|$ inside $\{0.25, 0.5, 0.75, 1\}$. To assess the performance of the methods, we measured their total execution time which breaks down to the time spent (i) to generate and sort the list of interval points L, denoted by Sorting, and (ii) to compute the *ICSJ* result, denoted by Joining.

4.2 Experiments

Figures 2 and 3 report the results of our experimental analysis. In specific, Figure 2 reports the total execution time of each method

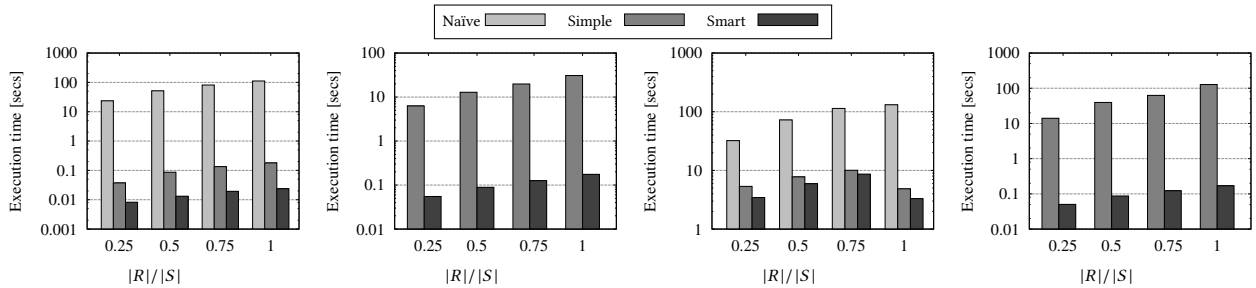

Figure 2: Total execution time while varying the $|R|/|S|$ ratio.

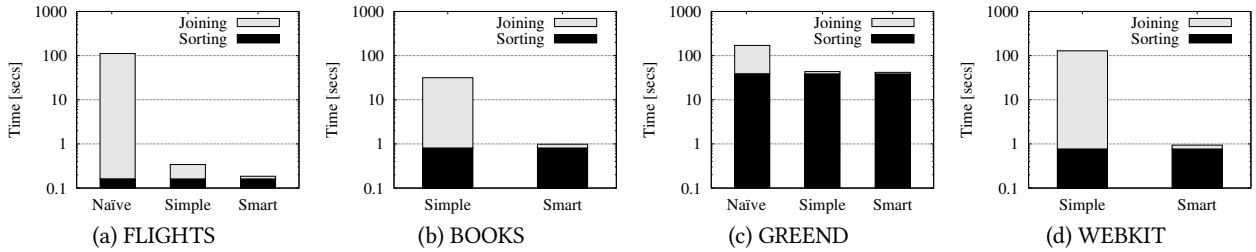

Figure 3: Execution time breakdown for $|R| = |S|$.

while varying the $|R|/|S|$ ratio and Figure 3 reports a breakdown of the execution time for the $|R| = |S|$ case. As expected, we were able to run the Naïve method only when $ICSJ$ was very selective, i.e., for datasets FLIGHTS and GREEND. Recall from Section 3 that Naïve first evaluates the IJ of the input collections; on BOOKS and WEBKIT, it was impossible to accommodate the enormous number of IJ result pairs in main memory.[4]

Figure 2 demonstrates the efficiency of the Smart Counting approach, which outperforms Simple Counting in all cases. In fact, Simple is competitive to Smart only for very selective join setups (see Figure 2(c)) while in all other cases, Smart is at least one order of magnitude faster. To explain the performance cost differences between Smart and Simple, we breakdown their execution times. In Figure 3(c), the execution cost of Simple is dominated by the generation and sorting of the points list L, because the number of overlapping interval pairs is small, rendering the inner loop at lines 19-20 of Algorithm 2 cheap. On other hand, when there is a large number of overlapping intervals, Figure 3 unveils that maintaining active sets and performing random accesses to update C counters severely impacts the joining time of Simple. In contrast, observe that the cost by Smart to compute the $ICSJ$ result is always much lower than that of generating/sorting L. This is expected because Smart is insensitive to the IJ result, as discussed in Section 3.2.

5 CONCLUSIONS

In this paper, we studied the evaluation of the interval count semi-join operation; we presented an efficient algorithm based on plane sweep. Our algorithm has lower complexity compared to state-of-the-art interval join algorithms if the number of join results is large. We experimentally showed that its overhead on top of sorting the data is minimal in all setups, which means that it is especially tailored in cases where the join inputs are already sorted (e.g., in streaming data applications). In the future, we plan to further study the semantics and the evaluation of top-k interval joins. We also intend to investigate the applications of interval joins and other temporal operations in streaming data.

[4]We also experimented with a version of Naïve that flushes the IJ result pairs on disk which was even slower.

ACKNOWLEDGEMENTS

This work was partially funded by the European Union's Horizon 2020 research and innovation programme under grant agreement No 657347.

REFERENCES

[1] Panagiotis Bouros and Nikos Mamoulis. 2017. A Forward Scan based Plane Sweep Algorithm for Parallel Interval Joins. *PVLDB* 10, 11 (2017), 1346–1357.
[2] Francesco Cafagna and Michael H. Böhlen. 2017. Disjoint interval partitioning. *VLDB J.* 26, 3 (2017), 447–466.
[3] Bhupesh Chawda, Himanshu Gupta, Sumit Negi, Tanveer A. Faruquie, L. Venkata Subramaniam, and Mukesh K. Mohania. 2014. Processing Interval Joins On Map-Reduce. In *EDBT*.
[4] Reynold Cheng, Sarvjeet Singh, Sunil Prabhakar, Rahul Shah, Jeffrey Scott Vitter, and Yuni Xia. 2006. Efficient join processing over uncertain data. In *CIKM*.
[5] Anton Dignös, Michael H. Böhlen, and Johann Gamper. 2014. Overlap interval partition join. In *SIGMOD*.
[6] Jost Enderle, Matthias Hampel, and Thomas Seidl. 2004. Joining Interval Data in Relational Databases. In *SIGMOD*.
[7] Jaewoo Kang, Jeffrey F. Naughton, and Stratis Viglas. 2003. Evaluating Window Joins over Unbounded Streams. In *ICDE*.
[8] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter M. Fischer, Donald Kossmann, Franz Färber, and Norman May. 2013. Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In *SIGMOD*.
[9] T. Y. Cliff Leung and Richard R. Muntz. 1992. Temporal Query Processing and Optimization in Multiprocessor Database Machines. In *VLDB*.
[10] Andrea Monacchi, Dominik Egarter, Wilfried Elmenreich, Salvatore D'Alessandro, and Andrea M. Tonello. 2014. GREEND: An energy consumption dataset of households in Italy and Austria. In *SmartGridComm*. 511–516.
[11] Jack A. Orenstein. 1986. Spatial Query Processing in an Object-Oriented Database System. In *SIGMOD*.
[12] Danila Piatov and Sven Helmer. 2017. Sweeping-Based Temporal Aggregation. In *SSTD*. 125–144.
[13] Danila Piatov, Sven Helmer, and Anton Dignös. 2016. An interval join optimized for modern hardware. In *ICDE*.
[14] Arie Segev and Himawan Gunadhi. 1989. Event-Join Optimization in Temporal Relational Databases. In *VLDB*.
[15] Cheng Sheng, Yufei Tao, and Jianzhong Li. 2012. Exact and approximate algorithms for the most connected vertex problem. *ACM TODS* 37, 2 (2012), 12:1–12:39.
[16] Manli Zhu, Dimitris Papadias, Jun Zhang, and Dik Lun Lee. 2005. Top-k Spatial Joins. *IEEE TKDE* 17, 4 (2005), 567–579.