Parallel In-Memory Evaluation of Spatial Joins

Dimitrios Tsitsigkos
Information Management Systems Institute
Athena RC, Athens, Greece
dtsitsigkos@imis.athena-innovation.gr

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

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

Manolis Terrovitis
Information Management Systems Institute
Athena RC, Athens, Greece
mter@imis.athena-innovation.gr

ABSTRACT

We study the in-memory and parallel evaluation of spatial joins, by tuning a classic partitioning based algorithm. Our study shows that, compared to a straightforward implementation of the algorithm, performance can be improved significantly. We also show how to select appropriate partitioning parameters based on data statistics, in order to tune the algorithm for the given join inputs. Our parallel implementation scales gracefully with the number of threads reducing the cost of the join to at most one second even for join inputs with tens of millions of rectangles.

CCS CONCEPTS

• **Information systems** → **Join algorithms**; **Spatial-temporal systems**; **Parallel and distributed DBMSs**;

KEYWORDS

Spatial Join, In-memory Data Management, Parallel Processing

ACM Reference Format:
Dimitrios Tsitsigkos, Panagiotis Bouros, Nikos Mamoulis, and Manolis Terrovitis. 2019. Parallel In-Memory Evaluation of Spatial Joins. In *27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '19), November 5–8, 2019, Chicago, IL, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3347146.3359343

1 INTRODUCTION

The spatial join is a well-studied fundamental operation. Given two collections of spatial objects R and S, the *spatial intersection join* returns all (r, s) pairs, such that $r \in R, s \in S$ and r and s have at least one common point. Due to the potentially complex geometry of the objects, intersection joins are typically processed in two steps. The *filter* step applies on spatial approximations of the objects, typically their *minimum bounding rectangle* (MBR). For each pair of object MBRs that intersect, the object geometries are fetched and compared in a *refinement step*. Similar to the vast majority of previous work [7], we focus on the filter step.

A wide range of spatial join algorithms have been proposed in the literature [3]. Given the fact that main memory chips become bigger and faster, in-memory join processing has recently received a lot of attendance [8]. In addition, given that commodity hardware supports parallel processing, multi-core join evaluation has also been the focus of recent research. Hence, in this paper, we target the parallel in-memory evaluation of spatial joins on modern hardware.

Our focus is the optimization of the simple, but powerful partitioning-based spatial join (PBSM) algorithm [9]. PBSM is shown to perform well in previous studies [8] and used by most distributed spatial data management systems [1, 6, 11]. In a nutshell, both datasets are first partitioned using a regular grid; each tile (cell) of the grid gets all rectangles that intersect it. Each tile defines a smaller spatial join task. These tasks are independent and can be executed in parallel, assigned to different threads or even to different machines in distributed evaluation. Typically a plane sweep algorithm based on forward scans [4] is used to process each task. For example, consider the two sets of MBRs of Figure 1a. Partitioning the rectangles using a 3×3 grid creates 9 independent spatial join tasks, one for each tile. Note that some rectangles may be replicated to multiple tiles. Because of this, some pairs of rectangles may be found to intersect in multiple tiles; e.g., r_1 intersects s_1 in tiles (0,0) and (0,1). Duplicate join results can be avoided by reporting a pair of rectangles only if a pre-determined reference point (typically, the top-left corner) of the intersection region is in the tile [5]; e.g., (r_1, s_1) is only reported by tile (0,0).

Currently, there is no comprehensive study so far on how the number and type of partitions should be defined. In this paper, we evaluate a 1D partitioning that divides the space into stripes (see Figure 1b), as opposed to the classic 2D partitioning, which uses a grid. Further, we investigate, for each partition, the best direction of the sweep line. Finally, we show how both the partitioning and the

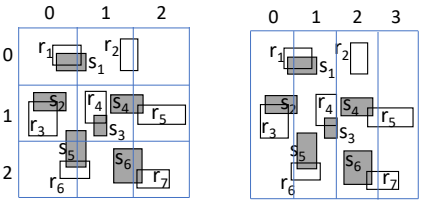


Figure 1: Example of PBSM: (a) 2D and (b) 1D partitioning

SIGSPATIAL '19, November 5–8, 2019, Chicago, IL, USA

ACM ISBN 978-1-4503-6909-1/19/11…$15.00
https://doi.org/10.1145/3347146.3359343

joining phases of the algorithm can be parallelized. Based on our tests, the 1D partitioning results in a more efficient algorithm. Also, increasing the number of partitions improves the performance of the algorithm, up to a point where adding more partitions starts having a negative effect. We present a number of empirical rules driven from data statistics (globally and locally for each partition) that can guide the selection of the algorithm's parameters. Finally, we evaluate the performance of the parallel version of the algorithm and show that it scales gracefully with the number of cores.

2 TUNING PBSM

PBSM is a popular spatial join algorithm for the following reasons:

- PBSM assumes no preprocessing or indexing of the data, so it can be applied on dynamically generated spatial data.
- The partitions define independent join tasks that can easily be distributed and/or parallelized.
- The number of join tasks is the same as the number of partitions.
- Producing duplicate results can be easily avoided.
- Implementing this approach is fairly easy.
- Previous studies [8] have shown that the performance of PBSM can hardly be beaten by more sophisticated approaches based on indexing or adaptive partitioning.

In the following, we explore the directions along which we can tune PBSM to improve its performance. We assume that PBSM uses the plane sweep algorithm of [4] for each partition-partition join.

2.1 One-dimensional Partitioning

The default partitioning approach for PBSM is a 2D grid, as shown in Figure 1a. Still, the same algorithm can be applied if we partition the data space in 1D *stripes*, as shown in Figure 1b. The stripes can be horizontal or vertical. Such a partitioning was considered by an *external memory plane sweep* join algorithm [2]; however, the objective of the partitioning there was to define the stripes in a way such that the "horizon" of the sweep line (which runs along the axis of the stripes) fits in memory. Since in this paper, we deal with in-memory joins, we do not consider this factor, but we study how the number of partitions affects the cost of the join.

2.2 Duplicate Elimination

Dittrich and Seeger [5] presented a simple but effective approach for eliminating duplicate results in PBSM. A rectangle pair is reported by a partition-partition join only if the top-left corner of their intersection area is inside the spatial extent of the partition. The pair of intersecting rectangles (r_1, s_1) in Figure 1a can be found in both tiles (0,0) and (0,1). However, this result will only be reported in tile (0,0), which contains the top-left corner of the intersection. Hence, for each rectangle pair found to intersect, a *duplicate* test is performed. Let $[r.x_l, r.x_u]$ and $[r.y_l, r.y_u]$ be the projections of rectangle r on the x and y axis, respectively. Let $[T.x_l, T.x_u]$ and $[T.y_l, T.y_u]$ be the corresponding projections of a tile. The duplicate test for pair (r, s), found to intersect in tile T, is the condition:

$$\max\{r.x_l, s.x_l\} \geq T.x_l \wedge \max\{r.y_l, s.y_l\} \geq T.y_l \quad (1)$$

Application to 1D partitioning. For the 1D partitioning, the duplicate test needs to apply a single comparison (as opposed to the two comparisons of Eq. 1). For example, if the stripes are vertical (as in Figure 1b), a join result is reported only if $\max\{r.x_l, s.x_l\} \geq T.x_l$.

2.3 Choosing the Sweeping Axis

When applying plane sweep for a tile (or stripe) T, we have to decide along which axis we will sort the rectangles and then sweep them. We devise a model which, given the sets of rectangles R_T, S_T inside a tile T, determines the sweeping axis to be used. The key idea is to estimate, for each axis, how many candidate pairs of rectangles from $R_T \times S_T$ intersect along this axis. For this purpose, we compute histogram statistics. In specific, we sub-divide the x and y projections of the tile T into a predefined number of partitions k. Then, we count how many rectangles from R and how many from S, x-intersect each x-division of the tile; the procedure for y partitions is symmetric. In this manner, we construct four histograms H_R^x, H_R^y, H_S^x, H_S^y of k buckets each. The number I_T^x of rectangles in $R_T \times S_T$ that x-intersect can then be approximated by accumulating the product of the corresponding histogram buckets, i.e.,

$$I_T^x = \sum_{i=0}^{k} \{H_R^x[i] \cdot H_S^x[i]\} \quad (2)$$

The smallest of I_T^x and I_T^y determines the chosen sweeping axis (i.e., x or y). For large tiles (compared to the size of the rectangles), we set $k = 1000$, while for small tiles k is the number of times the tile's extent is larger than the average rectangle extent. In practice, using all rectangles of T in the histogram construction is too expensive. So, we use a sample of rectangles from R_T and S_T for this purpose. Specifically, for every ϕ rectangles that are assigned to tile T, we use one for histogram construction. We set $\phi = 100$ by default because it can produce good enough estimates at a low overhead.

Application to 1D partitioning. Our model can be straightforwardly applied in case of a 1D partitioning; histogram statistics are now computed for the contents of the vertical or horizontal stripes and the entire domain on the other dimension.

3 PARALLEL PROCESSING

We parallelize evaluation by splitting its partitioning and joining phases into parallel and independent tasks, while trying to minimize the synchronization requirements between the threads. The steps for parallelizing the spatial join to m threads are as follows:

Partitioning phase

(1) Determine a division of each input R and S into m equi-sized parts arbitrarily.
(2) Initiate m threads. Thread i reads the i-th part of input R and *counts* how many rectangles should be assigned to each of the space partitions (tiles or stripes). Thread i repeats the same process for the i-th part of input S. Let $|R_T^i|$, $|S_T^i|$ be the numbers of rectangles counted by thread i for tile T and R, S, respectively.
(3) Compute $|R_T| = \sum_i^m |R_T^i|$ and $|S_T| = \sum_i^m |S_T^i|$ for each tile T. Allocate two memory segments for $|R_T|$ and $|S_T|$ rectangles of each partition T.
(4) Initiate m threads. Thread i reads the i-th parts of inputs R and S and partitions them. The memory allocated for each of $|R_T|$ and $|S_T|$ is logically divided into m segments based on the $|R_T^i|$'s and $|S_T^i|$'s. Hence, thread 1 will write to the first $|R_T^1|$ positions

Table 1: Datasets used in the experiments

source	dataset	alias	cardinality	avg. x-extent	avg. y-extent
Tiger 2015	AREAWATER	$T2$	2.3M	0.000007230	0.000022958
	EDGES	$T4$	70M	0.000006103	0.00001982
	LINEARWATER	$T5$	5.8M	0.000022243	0.000073195
	ROADS	$T8$	20M	0.000012538	0.000040672
OSM	Buildings	$O3$	115M	0.00000056	0.000000782
	Lakes	$O5$	8.4M	0.000021017	0.000028236
	Parks	$O6$	10M	0.000016544	0.000022294
	Roads	$O9$	72M	0.000010549	0.000016281

of $|R_T|$, thread 2 to the next $|R_T^2|$ positions, etc. After all threads complete partitioning, we will have the entire set of rectangles that fall in each tile continuously in memory.

Joining phase

(5) Construct two sorting tasks for each tile T (one for R_T and one for S_T). Assign the sorting tasks to the m threads.
(6) Construct a join task for each tile T (one for R_T and one for S_T). Assign the join tasks to the m threads.

Step 2 is applied to make proper memory allocation and prevent expensive dynamic allocations. It also facilitates the output of parallel partitioning for each tile T to be continuous in memory during Step 4. When the model of Section 2.3 is used, the histograms are computed while loading input data (i.e., in either of Steps 2 and 4).

4 EXPERIMENTAL ANALYSIS

4.1 Setup

We experimented with Tiger 2015 and OpenStreetMap (OSM) datasets from [6].[1] For each dataset, we computed the MBRs of the objects and came up with a corresponding collection of rectangles. Table 1 details the datasets we used. Dataset cardinality ranges from 2.3M to 115M objects and we tested joins having inputs from the same collection, with similar or various scales. The last two columns of the tables are the relative (over the entire space) average length of the rectangle projections at each axis.

We implemented the spatial join algorithm (all different versions) in C++ and compiled it using gcc (v4.8.5). For multi-threading, we used OpenMP. All experiments were run on a machine with 384 GBs of RAM and a dual 10-core Intel(R) Xeon(R) CPU E5-2630 v4 clocked at 2.20GHz running CentOS Linux 7.3.1611; with hyper-threading, we were able to run up to 40 threads. The reported runtimes include the costs of partitioning both datasets and then joining them. Due to lack of space, our full set of experiments can be found in [10].

4.2 Selecting the Sweeping Axis

We first test the effect that the sweeping axis selection (either x or y) has on the performance of the algorithm. For this purpose, we chose not to partition the data, but ran the single-threaded plane-sweep join from [4] in the entire dataspace (i.e., modeling the case of a single tile). Table 2 reports the execution times per query. We observe that sweeping along the wrong axis may even double the cost of the join. The last column of the table reports the result of running our model (Eq. 2). Our model was able to accurately determine the proper sweeping axis in all cases. Note that the cost of this decision-making process is negligible compared to the partitioning and joining cost; even for the largest queries, our model needs less than 10 milliseconds.

[1] http://spatialhadoop.cs.umn.edu/datasets.html

Table 2: Sweeping axis effect; queries ordered by runtime

query	sweeping axis		adaptive model	
	x	y	I^x	I^y
$T2 \bowtie T5$	8.94s	16.96s	8,376	19,232
$T2 \bowtie T8$	24.52s	40.72s	8,895	18,660
$O5 \bowtie O6$	24.92s	66.06s	2,692	12,279
$O6 \bowtie O9$	216.88s	444.19s	3,989	11,510
$T4 \bowtie T8$	674.50s	1,360.92s	8,135	19,406
$O9 \bowtie O3$	926.14s	1,681.30s	4,535	11,529

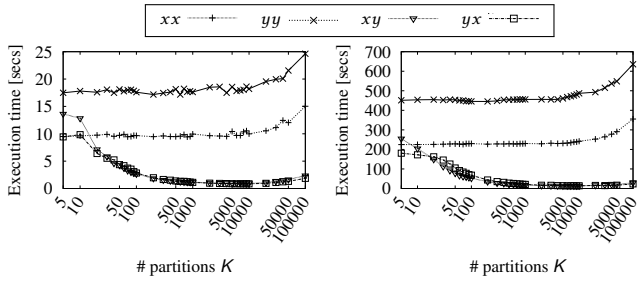
4.3 Evaluation of Partitioning

Next, we investigate the impact of partitioning to the performance of the algorithm. We tune 1D and 2D-based PBSM and then compare the two partitioning approaches to each other.

Tuning 1D Partitioning. Figure 2 reports the cost of two spatial join queries while varying the number K of (uniform) 1D partitions. We tested all combinations of partitioning and sweeping axes. For example, xy denotes partitioning along the x axis (to vertical stripes) and sweeping along the y axis. Note that if the sweeping axis is the same as the partitioning axis (i.e., cases xx and yy), the join cost does not drop when we increase the number of partitions K. This is expected because, regardless the number of partitions, case xx or yy is equivalent to having no partitions at all and sweeping along the x or y axis in the entire space. When K is too large, the costs of xx and yy increase because the partitions become very narrow and replication becomes excessive. On the other hand, the performance of cases xy and yx improves with K and, after some point, i.e., K = 2,000, they converge to the same (very low) cost. The costs of both xy and yx start to increase again when K > 10,000, at which point we start having significant replication (observe the average x- and y-extent statistics in Table 1). Figure 3 breaks down the total cost to partitioning and joining for the xy case. The joining cost includes the cost of sorting the partitions. As expected, the cost of partitioning increases with K and the joining cost drops. After K = 10,000 partitioning becomes very expensive without offering improvement in the join. The lowest runtime is achieved when the x-extent of the partitions (i.e., the narrow side of the stripes) is about 10 times larger than the average x-extent of the rectangles.

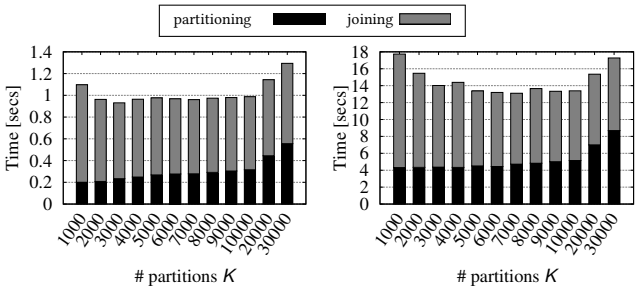
Tuning 2D Partitioning. We vary the granularity $K \times K$ of the grid and measure for each value of K the runtime cost of the algorithm, when the sweeping axis is always set to x, always set to y, or when our adaptive model is used to select the sweeping axis at each tile (which could be different at different tiles). Figure 4 depicts the performance of the three join variants. Similarly to 1D partitioning, when the number of partitions is small $K \leq 20$, the choice of the sweeping axis makes a difference and choosing x is better. In these configurations, our model can be even better than always choosing x. The three options converge at about K = 500. Figure 5 shows the cost breakdown for the partitioning and joining phases of the 2D spatial join, when our model is used for picking the sweeping axis x. The observations are similar the corresponding ones for 1D partitioning. The best grid configuration is around K = 2,000, which is consistent with the best option in 1D partitioning.

1D vs. 2D Partitioning. There are two main findings from the PBSM tuning experiments. First, the rule of the thumb is to select K (in both 1D and 2D partitioning) such that the extents of the resulting partitions are about one order of magnitude larger than the extents of the rectangles (in one or both dimensions, respectively).



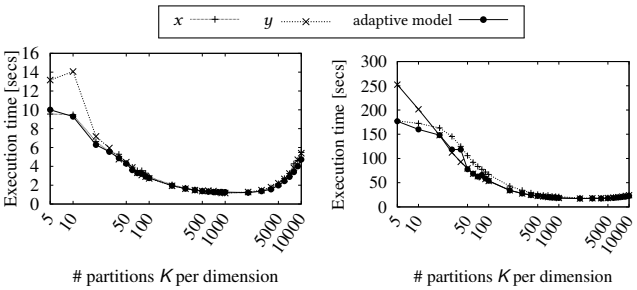
(a) $T2 \bowtie T5$ (b) $O6 \bowtie O9$

Figure 2: Tuning 1D partitioning: total execution time



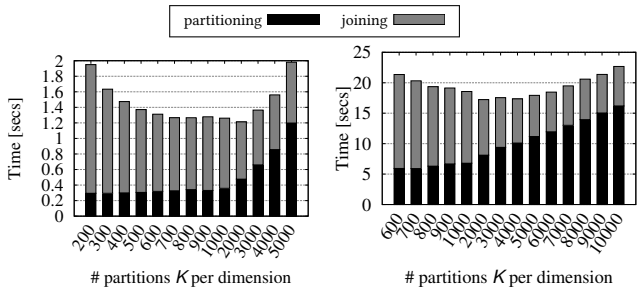
(a) $T2 \bowtie T5$ (b) $O6 \bowtie O9$

Figure 3: Tuning 1D partitioning: time breakdown



(a) $T2 \bowtie T5$ (b) $O6 \bowtie O9$

Figure 4: Tuning 2D partitioning: total execution time



(a) $T2 \bowtie T5$ (b) $O6 \bowtie O9$

Figure 5: Tuning 2D partitioning: time breakdown

Table 3: 1D vs. 2D partitioning: speedup

query	1D		2D	
	K	speedup	$K \times K$	speedup
$T2 \bowtie T5$	3000	9.6x	1000×1000	8.16x
$T2 \bowtie T8$	7000	10.67x	2000×2000	8.98x
$O5 \bowtie O6$	3000	8.62x	1000×1000	6.82x
$O6 \bowtie O9$	7000	16.56x	2000×2000	12.58x

Table 4: Parallel evaluation: runtime (1D partitioning)

# threads	queries			
	$O5 \bowtie O6$	$O6 \bowtie O9$	$T4 \bowtie T8$	$O9 \bowtie O3$
1	2.98s	14.4s	20.1s	43.0s
5	0.75s	3.32s	4.34s	10.6s
10	0.46s	1.91s	2.47s	6.11s
15	0.38s	1.45s	1.85s	4.54s
20	0.32s	1.21s	1.64s	3.54s
25	0.29s	1.07s	1.42s	3.09s
30	0.28s	0.99s	1.36s	2.89s
35	0.27s	0.96s	1.27s	2.72s
40	0.27s	0.91s	1.21s	2.72s

Second, 1D partitioning achieves better performance compared to 2D partitioning, due to less replication and the fact that all tiles in a row or a column can be swept by a single line (along the row or column) with the same effect as processing all tiles independently with sweeping along the same direction. Table 3 summarizes, for the four join queries, the best speedups achieved by 1D and 2D partitioning, compared to the best corresponding performance of the plane sweep algorithm without partitioning. 1D partitioning is up to 32% faster compared to 2D partitioning.

4.4 Parallel Evaluation

Last, we test the parallel version of the algorithm using 1D partitioning. Table 4 summarizes, for the four join queries, the runtime achieved by our parallel evaluation. The performance scales gracefully with the number of threads, until it stabilizes over 20 threads, which equals the number of physical cores in our machine. As a general conclusion, our parallel design takes full advantage of the system resources to greatly reduce the join cost.

ACKNOWLEDGEMENTS

Partially supported by the project "Moving from Big Data Management to Data Science" (MIS 5002437/3) co-financed by Greece and the European Union (European Regional Development Fund).

REFERENCES

[1] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel H. Saltz. 2013. Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce. *PVLDB* 6, 11 (2013), 1009–1020.
[2] Lars Arge, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel, and Jeffrey Scott Vitter. 1998. Scalable Sweeping-Based Spatial Join. In *VLDB*. 570–581.
[3] Panagiotis Bouros and Nikos Mamoulis. 2019. Spatial Joins: What's next? *SIGSPATIAL Special* 11, 1 (2019), 13–21.
[4] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. 1993. Efficient Processing of Spatial Joins Using R-Trees. In *SIGMOD Conference*. 237–246.
[5] Jens-Peter Dittrich and Bernhard Seeger. 2000. Data Redundancy and Duplicate Detection in Spatial Join Processing. In *ICDE*. 535–546.
[6] Ahmed Eldawy and Mohamed F. Mokbel. 2015. SpatialHadoop: A MapReduce framework for spatial data. In *ICDE*. 1352–1363.
[7] Edwin H. Jacox and Hanan Samet. 2007. Spatial join techniques. *ACM Transactions on Database Systems* 32, 1 (2007), 7.
[8] Sadegh Nobari, Qiang Qu, and Christian S. Jensen. 2017. In-Memory Spatial Join: The Data Matters!. In *EDBT*. 462–465.
[9] Jignesh M. Patel and David J. DeWitt. 1996. Partition Based Spatial-Merge Join. In *SIGMOD Conference*. 259–270.
[10] Dimitrios Tsitsigkos, Panagiotis Bouros, Nikos Mamoulis, and Manolis Terrovitis. 2019. Parallel In-Memory Evaluation of Spatial Joins. (2019). arXiv:1908.11740
[11] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Zhiyong Xu. 2009. SJMR: Parallelizing spatial join with MapReduce on clusters. In *CLUSTER*. 1–8.