Spatial Joins in Main Memory: Implementation Matters!

Darius Šidlauskas
Aarhus University
dariuss@cs.au.dk

Christian S. Jensen
Aalborg University
csj@cs.aau.dk

ABSTRACT

A recent PVLDB paper reports on experimental analyses of ten spatial join techniques in main memory. We build on this comprehensive study to raise awareness of the fact that empirical running time performance findings in main-memory settings are results of not only the algorithms and data structures employed, but also their implementation, which complicates the interpretation of the results.

In particular, we re-implement the worst performing technique without changing the underlying high-level algorithm, and we then offer evidence that the resulting re-implementation is capable of outperforming all the other techniques. This study demonstrates that in main memory, where no time-consuming I/O can mask variations in implementation, implementation details are very important; and it offers a concrete illustration of how it is difficult to make conclusions from empirical running time performance findings in main-memory settings about data structures and algorithms studied.

1. INTRODUCTION

Stated briefly, a *spatial join* takes two spatial data sets as arguments and returns pairs of objects from the two sets that have intersecting spatial extents. Spatial joins are important in a range of applications.

Further, efficient *in-memory* spatial join processing is important because of the continuously decreasing prices and increasing capacities of RAM chips, which enable single-server machines with terabytes of main-memory storage, and because the size of raw location data is often tiny. For example, two dimensional coordinates are often encoded as two 4-byte single-precision or integer values [2, 3, 7, 8].

Sowel et al. [7] report on a thorough experimental performance study of ten spatial join techniques in main memory. The techniques are first optimized for in-memory performance and then studied in the same framework.

The techniques are divided into four categories, among which we focus on the *static index nested loop join* category. Here, a static index is built, and index nested loops are used to compute join results. The following four static indexes belong to this category: R-Tree [4, 6], CR-Tree [5], Linearized KD-Trie [3], and Static Grid [8]. This category reports the best performance results on average [7].

In Section 2, we repeat the experiments for the static indexes using the source code of the experimental framework employed in the original study [1]. The results match the results reported in the original paper: The static, uniform grid-based technique, termed *Simple Grid*, exhibits the worst performance by a large margin and in all cases.

In Section 3, we analyze the implementation of Simple Grid and make simple modifications to its grid directory, bucket list, and algorithm implementations. We report the performance improvements gained from each modification. In combination, they yield approximately a 6-fold improvement in performance and elevate Simple Grid from worst to best.

2. REPRODUCIBILITY STUDY

We describe the experimental setup and then report the results of the reproducibility study.

2.1 Experimental Setup

The hardware used is a quad-core Intel i7 CPU clocked at 3.4 GHz with 16 GB of RAM.

The source code of the experimental framework [1] contains the implementations of the ten spatial join techniques considered as well as the workload generator used [7]. Due to the 4-page limit, we show results only for synthetic workloads (based on [2]), but the same performance trends also hold for the simulation workloads.

Workload parameters are given in Table 1 (with default values in bold). A two-dimensional setting is assumed where the processing is modeled using discrete time-steps called *ticks*. Each tick consists of two non-overlapping phases: query and update. In the query phase, some fraction of spatial objects (% Queriers) issues spatial queries, which are rectangular range queries. In the update phase, some fraction of objects (% Updaters) issues updates. Each update may change an object's velocity or position, and different

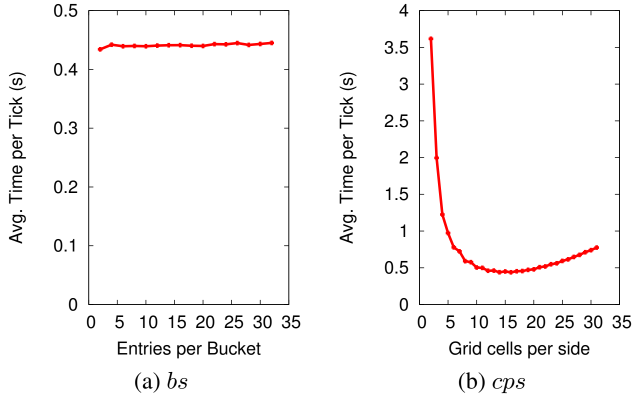
Parameter	Uniform	Gaussian
Number of Ticks	100	120
Number of Points	10K .. **50K** .. 90K	50K
Space Size	$10K^2$ .. $\mathbf{22K^2}$ .. $30K^2$	$22K^2$
Maximum Speed	200	200
Query Size	400	400
% Queriers	10% .. **50%** .. 90%	50%
% Updaters	10% .. **50%** .. 90%	N/A

Table 1: Parameters for Synthetic Workloads

join algorithms handle updates differently. Objects can only read the state of other objects as of the previous tick, and all updates are applied at the end of each tick.

In the uniform workloads, objects are placed at random locations in the data space, and their speeds and directions are chosen at random. In the Gaussian workload, objects are placed around a fixed set of hotspots, and their movements follow a Gaussian-like distribution. For the complete description of the experimental setup, we refer to the original study [7].

Optimal parameter values for all competing spatial join approaches are determined with parameter sweeps [7]. We repeat the same experiments to determine the optimal parameters on our hardware. For example, to determine the optimal values of the bucket size (bs) and the number of cells per side (cps) for Simple Grid, the values are varied from 4 to 32 and the corresponding performance is measured. The results are shown in Figure 1. We obtain the same optimal configuration for Simple Grid, i.e., $bs = 4$ and $cps = 13$, as in the original study. Varying bs has no effect on performance, while



(a) bs (b) cps

Figure 1: Tuning Original Simple Grid

varying cps significantly affects the performance, with the best configuration being a relatively coarse-granularity grid (13×13). We obtained the same or very similar optimal configurations as reported in [7] for all techniques.

2.2 Results

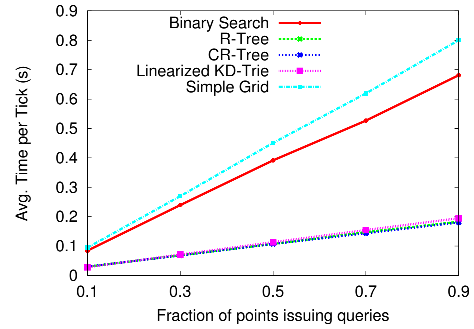
Figure 2 shows the performance of all the techniques that use different static indexes. For each technique, the performance trends are very similar to those originally reported, although the results are slightly better due to our faster machine (3.4 versus 2.66 GHz).

The uniform grid-based approach, Simple Grid, performs the worst in all cases. It falls behind even a baseline approach, termed *Binary Search*, where the data points are sorted by one coordinate, upon which a nested loop with binary search (on the sorted coordinate) is used to compute the join. There is no clear winner: the best-performing approaches, namely R-Tree [4, 6], CR-Tree [5], and Linearized KD-Trie [3], exhibit very similar performance.

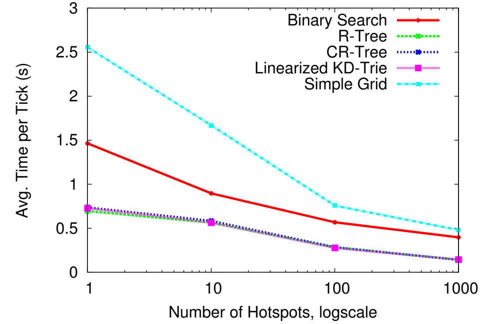
Next, as in the original study, we break the performance into build, query, and update execution. Table 2 shows the average times in seconds per tick (for the moment, ignore the lower part of the table). Again, we observe the same performance trends, albeit the run times are shorter on our faster machine.

3. SIMPLE GRID

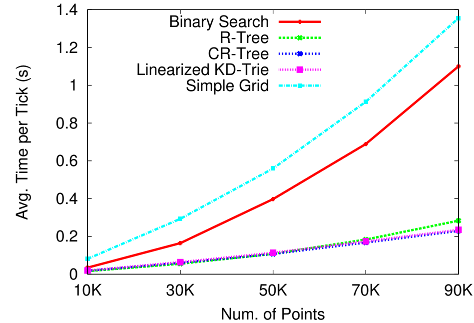
We proceed to improve the implementation of the worst-performing technique, Simple Grid, which is described as follows [7]:



(a) Scaling the Query Rate



(b) Scaling the Number of Hotspots



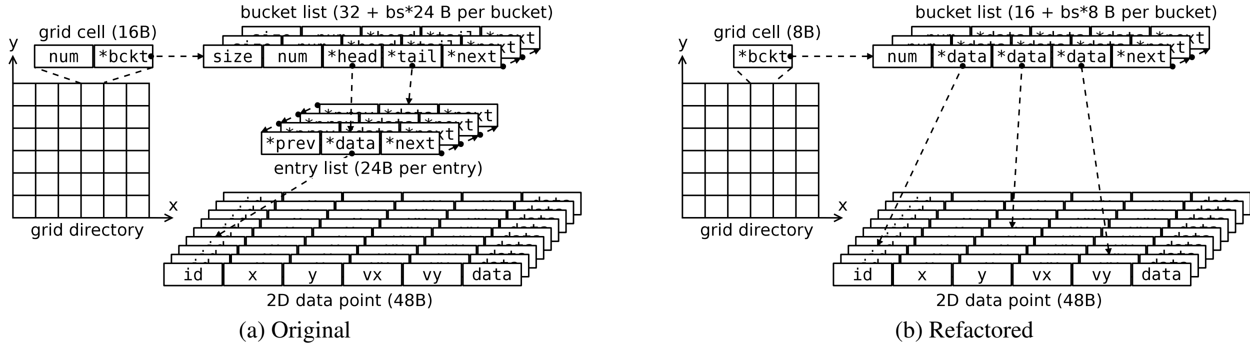
(c) Scaling the Number of Points

Figure 2: Reproduced Performance of Static Indices

Method	Build (s)	Query (s)	Update (s)
R-Tree	0.008	0.098	0.0012
CR-Tree	0.009	0.096	0.0009
Lin. KD-Trie	0.005	0.107	0.0008
Simple Grid	0.0019	0.559	0.0029
+restructured	0.0015	0.494	0.0022
+querying	0.0015	0.381	0.0023
+bs tuned	0.0007	0.277	0.0009
+cps tuned	0.0009	0.093	0.0010

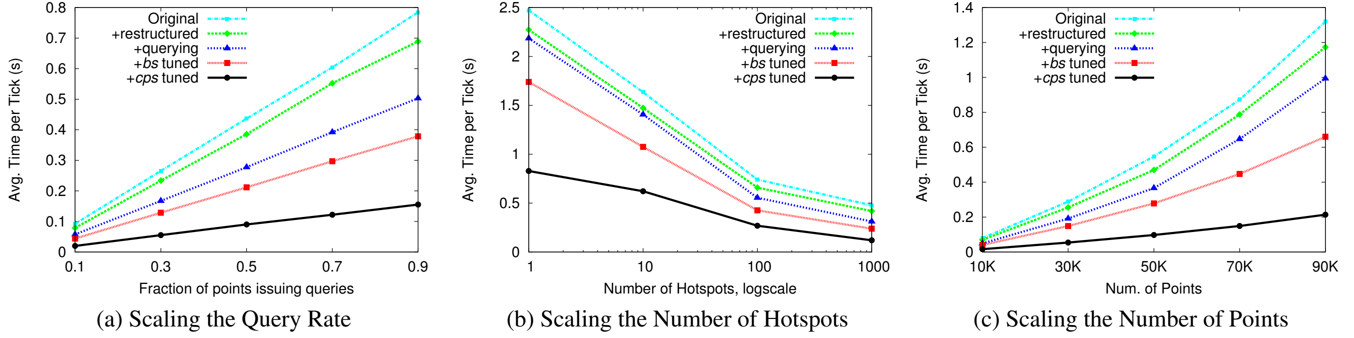
Table 2: Breakdown: 50% queries and updates, 50K points

This index partitions space uniformly into a fixed number of cells stored as a two-dimensional array. Each cell contains a pointer to a linked list of buckets storing the points that fall within that cell. The search algorithm must examine every cell that intersects the query region.



(a) Original (b) Refactored

Figure 3: Simple Grid Structure



(a) Scaling the Query Rate (b) Scaling the Number of Hotspots (c) Scaling the Number of Points

Figure 4: New Performance of Simple Grid

This description clearly defines the main concepts behind the uniform grid-based technique. At the implementation level, needless to say, there are plenty of ways to realize this technique. In Sections 3.1 and 3.2, we analyze the actual index structure and query algorithm implementations of Simple Grid, proposing and evaluating improvements (based on [8]). In Section 3.3, we re-calibrate the improved implementation and report the performance effects of each improvement.

3.1 Structure

Figure 3a depicts the structure of Simple Grid as implemented in [1]. Two implementation details are important. First, the grid directory is implemented as a contiguous array of (integer, pointer) pairs. The integer is used for counting the number of objects stored in a cell, while the pointer references the singly-linked list of buckets that store the cell's data entries. As such, 16 bytes are required per grid cell.

Second, each bucket contains a doubly-linked list of pointers to the actual data entries. Such a list requires 24 bytes per data point and is stored in a bucket that requires 32 bytes (cf. Figure 3a). Assuming bs is the bucket size and n is the number of objects, the memory consumption is $n \times 24 + n/bs \times 32 = n(24 + 32/bs)$. Next, the best-performing bucket size is 4 (as found by [7] and us). This implies that in addition to the grid directory storage, Simple Grid consumes 32 bytes extra for each indexed object.

We propose two simple modifications to the structure of Simple Grid, as shown in Figure 3b. First, we remove the unnecessary integer and store only a pointer in each grid cell. This cuts the grid directory's memory requirements by half. Second, we remove the doubly-linked lists of data pointers and store them directly in the buckets. The resulting memory consumption is $n \times 8 + n/bs \times 16 = n(8 + 16/bs)$. Assuming the same configuration ($bs = 4$), Simple Grid requires only 12 bytes per point.

These modifications are crucial in main memory. In addition to reduced memory footprint, Simple Grid packs more data per cache line and thus improves its cache hit rate. Also, the restructured design removes an unnecessary layer of indirection, implying that reaching each point's data requires one hop less and thus avoids another potential cache miss.

It is possible to further improve locality by storing the spatial attributes (the x and y coordinates) in the buckets, too. However, as the targeted setting assumes that secondary indexes are used (the algorithms operate on pointers and never update the base data directly), we do not introduce this modification. All the techniques studied comply with this assumption [1].

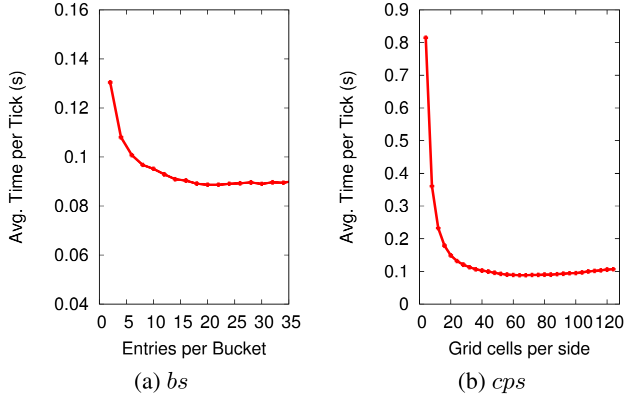
In Figure 4, the lines labeled "+restructured" show the performance of Simple Grid after the above changes are applied. While the performance gain is minor (a 1.13-fold speedup under the default workload), we expect to benefit from an increased bs, as larger buckets improve data locality.

3.2 Query Algorithm

Algorithm 1 illustrates Simple Grid's original range query algorithm. Given the input region r, defined by its lower-left (x_1, y_1) and upper-right (x_2, y_2) corners, the algorithm traverses *all* grid cells one by one. If the current cell is fully contained in r, all its points are reported. If the current cell only intersects with r, each point is checked individually for containment in r.

We refactor the algorithm as shown in Algorithm 2. Instead of scanning the entire grid directory, we compute and scan only the

overlapping cells. In Figure 4, the lines labeled "+querying" show the performance of Simple Grid with these changes. We get another 1.3-fold speedup and also expect to benefit from more granular grid cells.



(a) *bs* (b) *cps*

Figure 5: Tuning Refactored Simple Grid

Algorithm 1: `enumerateOverRegion`(Region2D r)

1. *cps* ← configured based on Figure 1b;
2. **for** $i = 1$ **to** *cps* **do**
3.   **for** $j = 1$ **to** *cps* **do**
4.   GridCell *cell* = `getCell` (i, j) ;
5.   **if** *r contains cell* **then**
6.   **foreach** $p \in cell$ **do**
7.   reportPoint(p);
8.   **else if** *r intersects cell* **then**
9.   **foreach** $p \in cell$ **and** $p \in r$ **do**
10.   reportPoint(p);

Algorithm 2: `enumerateOverRegion`(Region2D r)

1. $cellSize$ ← set based on *cps*;
2. $x_{min} = r.x_1/cellSize$; $x_{max} = r.x_2/cellSize$;
3. $y_{min} = r.y_1/cellSize$; $y_{max} = r.y_2/cellSize$;
4. **for** $i = x_{min}$ **to** x_{max} **do**
5.   **for** $j = y_{min}$ **to** y_{max} **do**
6.   Lines 4–10 in Algorithm 1;

3.3 Parameter Tuning

Having applied the above modifications, we rerun the experiments where *bs* and *cps* are varied. The results are shown in Figure 5. As expected, larger values for both *bs* and *cps* are preferred: the values that yield the best performance are 20 and 64, respectively. The larger *bs* makes it possible to exploit data locality in a bucket, while the more granular grid (larger *cps*) better adapts to the range queries in the workloads.

In Figure 4, the lines labeled "+*bs* tuned" show the performance of Simple Grid after it is configured with $bs = 20$. The reconfiguration improves the performance 1.4-fold. The lines labeled "+*cps* tuned" show the performance after Simple Grid is further tuned with $cps = 64$. This causes an additional 3-fold improvement under the default workload. As a result, Simple Grid becomes the top performer among all techniques (see "+*cps* tuned" in Table 2).

Finally, Table 3 shows the cycles-per-instruction (CPI), the number of total instructions (in billions), and the number of cache misses (in millions) at different levels of the memory hierarchy in Simple Grid before and after our modifications. We observe huge improvements across all measurements.

Simple Grid	CPI	Total INS	Data Cache Misses		
			L1	L2	L3
Before	1.32	171 B	8,786 M	6,148 M	325 M
After	1.13	37 B	1,091 M	747 M	67 M

Table 3: Profiling: 50% queries and updates, 50K points

4. CONCLUSIONS

The results reported here suggest that, in the setting considered, substantial performance gains can be achieved by means of careful implementation, to the point that the implementations of the data structures and algorithms are more important for the performance than the data structures and algorithms themselves. In addition, the results illustrate how empirical performance findings, which are artifacts of not only the high-level data structures and algorithms considered, but also their implementation, renders it challenging to conclude about the data structures and algorithms from such findings.

More generally, the results suggest that the empirical study of algorithms and data structures in main-memory settings, even single-threaded settings, is much more challenging than in disk-based settings.

Acknowledgments We thank the reviewers for their helpful comments. The research was supported in part by the Danish National Research Foundation grant DNRF84 through Center for Massive Data Algorithmics (MADALGO) and by a grant from the Obel Family Foundation.

5. REFERENCES

[1] Spatial indexing at Cornell. http://www.cs.cornell.edu/bigreddata/spatial-indexing/. Accessed: 2014-07-23.

[2] S. Chen, C. S. Jensen, and D. Lin. A benchmark for evaluating moving object indexes. *PVLDB*, 1(2):1574–1585, 2008.

[3] J. Dittrich, L. Blunschi, and M. A. V. Salles. Indexing moving objects using short-lived throwaway indexes. In *SSTD*, pages 189–207, 2009.

[4] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[5] K. Kim, S. K. Cha, and K. Kwon. Optimizing multidimensional index trees for main memory access. In *SIGMOD*, pages 139–150, 2001.

[6] S. T. Leutenegger, M. A. Lopez, and J. Edgington. STR: A simple and efficient algorithm for R-tree packing. In *ICDE*, pages 497–506, 1997.

[7] B. Sowell, M. V. Salles, T. Cao, A. Demers, and J. Gehrke. An experimental analysis of iterated spatial joins in main memory. *PVLDB*, 6(14):1882–1893, 2013.

[8] D. Šidlauskas, S. Šaltenis, C. W. Christiansen, J. M. Johansen, and D. Šaulys. Trees or grids? Indexing moving objects in main memory. In *GIS*, pages 236–245, 2009.