reTOUCH: A more workload balanced In-Memory Spatial Join by Iterative Hierarchical Data-Oriented Partitioning

ABSTRACT

Efficient spatial joins are pivotal for many applications and particularly important for geographical information systems or for the simulation sciences where scientists work with spatial models. Past research has primarily focused on disk-based spatial joins; efficient in-memory approaches, however, are important for two reasons: a) main memory has grown so large that many datasets fit in it and b) the in-memory join is a very time-consuming part of all disk-based spatial joins.

In this paper we develop **TOUCH**, a novel in-memory spatial join algorithm that uses hierarchical data-oriented space partitioning, thereby keeping both its memory footprint and the number of comparisons low. Our results show that TOUCH outperforms known in-memory spatial-join algorithms as well as in-memory implementations of disk-based join approaches. In particular, it has a one order of magnitude advantage over the memory-demanding state of the art in terms of number of comparisons (i.e., pairwise object comparisons), as well as execution time, while it is two orders of magnitude faster when compared to approaches with a similar memory footprint. Furthermore, TOUCH is more scalable than competing approaches as data density grows.

Categories and Subject Descriptors

H.2.8 [**DATABASE MANAGEMENT**]: Spatial databases and GIS; G.2.2 [**DISCRETE MATHEMATICS**]: Graph Theory

Keywords

Scalable algorithms; Spatial joins; TOUCH; Indexing

1. INTRODUCTION

Many applications dealing with spatial data rely on the efficient execution of spatial or distance joins. In geographical applications these joins are used to detect collisions or proximity between geographical features [30], i.e., landmarks, houses, roads, etc. and in medical imaging, spatial joins are used to determine which cancerous cells are within a certain distance of each other [10]. In the simulation sciences, where scientists build and simulate precise spatial models of the phenomena they are studying, distance joins are, for example, used to monitor the folding process of peptides [12].

Unfortunately, a distance join on two unsorted and unindexed datasets is a computationally costly operation, even if executed in the main memory of a supercomputer. Data models grow fast and, as a result, the distance join is a bottleneck in many scientific applications today, preventing them from scaling to bigger models. Apart from growing, data models in real-world scientific applications also become increasingly realistic and hence denser. The growing density of the models substantially increases the join's selectivity, rendering the efficient execution of this operation key for scaling to larger and more realistic models.

To formulate the problem, we translate the distance join into a spatial join that tests pairs of objects for intersection. Formally, the distance join takes a distance ϵ and two spatial datasets A and B and finds all pairs of spatial objects $a \in A$ and $b \in B$ such that the distance between a and b is less than or equal to ϵ. To translate the problem into a spatial join, we increase the size of all objects in one dataset by ϵ and then test both datasets for intersecting objects [15].

Existing research on spatial joins has mostly focused on disk-based approaches. Spatial join techniques designed for use in memory hence lack efficiency and scalability. In this paper we develop TOUCH, a two-way spatial join approach that works efficiently in memory. TOUCH combines concepts from previous work and avoids their problems, i.e., excessive memory footprint and excessive number of comparisons. In particular, it uses a hierarchy to mitigate replication of elements and data-oriented partitioning to avoid excessive pairwise comparisons. Additionally, data-oriented partitioning further reduces the number of comparisons by not considering the objects spatially far from other objects.

Because of the limited work on in-memory spatial joins, we also draw inspiration from on-disk approaches and compare our approach to disk-based approaches used in memory. The latter is reasonable as growing memory capacities allow for approaches with a bigger memory footprint, originally designed for use on disk, to be used in main memory.

We apply our solution on the "touch detection" problem, a challenging neuroscience application that arises in collaboration with computational neuroscientists. The neuroscientists build biophysically realistic models with data acquired during anatomical research of the rat brain. In their models, as in the rat brain, each neuron has branches extending into large parts of the tissue. The neurons receive and send information to other neurons using these branches. To de-

termine where in the model to place the synapses (structure that permits a neuron to pass a signal to another neuron), it suffices to find the places where the distance between two branches is below a given threshold [18], i.e., where the neurons touch.

Our experiments show that TOUCH outperforms existing spatial joins algorithms in terms of number of comparisons and execution time. When compared to the fastest related approach TOUCH requires substantially less memory. In the context of our working example, TOUCH performs the spatial join at least one order of magnitude faster than related work. Our experiments also indicate that TOUCH will scale better to more detailed and denser neuroscience datasets in the future.

Our work has significant potential impact beyond neuroscience applications. Spatial joins are broadly used in many applications beyond neuroscience and the simulation sciences, and their use in memory is becoming increasingly important in many applications for two reasons. First, main memory has grown so large that many datasets fit into it directly and the spatial join can be entirely performed in memory. Second, the in-memory join is also an integral part of all disk-based joins. Disk-based joins partition datasets that do not fit in memory and then join the partitions in memory. Speeding up the in-memory join helps to considerably speed up on-disk approaches as a whole as well, particularly if the join is very selective and many objects need to be compared (thus spending a large share of the overall time for the in-memory join).

The remainder of the paper is structured as follows. We discuss related work and its shortcomings in Section 2. In Section 3 we motivate our work and in Section 4 we present TOUCH, our algorithm, and discuss its implementation in Section 6. We compare TOUCH to related approaches in Section 7 and draw conclusions in Section 8.

2. RELATED WORK

While several spatial join approaches have been developed for disk in the past, only few have been developed for use in memory. Many disk-based spatial join algorithms, however, can also be used in memory. In the following we discuss all related work no matter if it has traditionally been used for disk or in memory.

2.1 In-Memory Approaches

Only two approaches have been developed for use in memory and thus have a small memory footprint: the nested loop join [24] and the plane-sweep join [28].

The nested loop join iterates over both spatial datasets in a nested loop and compares all pairs of objects. While this results in a complexity of $O(n^2)$, no additional data structures are needed, making the approach very space efficient.

The plane-sweep approach sorts the datasets in one dimension and scans both datasets synchronously. All objects on the sweep plane (stored in an efficient data structure) are compared to each other to see if they overlap. Because the objects are only sorted in one dimension, objects which are not near each other in the other dimensions may be on the sweep plane at the same time, thus leading to redundant comparisons and hence slowing down the approach.

Despite its deficiencies, however, the plane-sweep approach is still broadly used to join in memory the partitions resulting from disk-based spatial joins.

2.2 On-disk Approaches

Because they can also be used in memory, in the following we discuss distance join approaches designed for disk. We categorize the approaches based on whether they require an index on both, one, or none of the datasets.

2.2.1 Both Datasets Indexed

If both datasets A and B are indexed with R-Trees [13], a synchronous traversal [7] can be used to join them. With the R-Tree indexes I_A and I_B on datasets A and B, this approach starts from the roots of the trees and synchronously traverses the tree to the leaf level. If two nodes $n_A \in I_A$ and $n_B \in I_B$ on the same level (one from each tree) intersect, then the children of n_A will be tested for intersection with the children of n_B. This process recursively traverses the trees to the leaf level where the objects are compared.

By building on the R-Tree, this approach also inherits the problems of the R-Tree, namely inner node overlap and dead space. Overlap in the R-Tree structure leads to too many comparisons and hence slows down the join operation. Extensions like the R*-Tree [6] or the R+-Tree [29] have been proposed to reduce overlap. The former tackles overlap with an improved node split algorithm (reinsertion of spatial objects if a node overflows) while the latter duplicates objects to reduce overlap. Duplicating objects, however, also leads to duplicate results which have to be filtered.

Arguably the most efficient R-Trees can be built through bulkloading if the data is known a priori. Several bulokloading approaches like the STR [19], Hilbert [16], TGS [11] and the PR-Tree [4] have been developed, all yielding better performance than R+-Tree or R*-Tree. The R-Tree resulting from bulkloading with Hilbert and STR perform similarly and outperform TGS as well as the PR-Tree on real-world data. TGS and the PR-Tree, however, outperform STR and Hilbert on data sets with extreme skew and aspect ratio.

Double index traversals are also possible with Quadtrees [2] (or Octrees in 3D). Similar to the R+-Tree objects are duplicated (or references to the objects) and duplicate results are possible and need to be filtered at the end [3].

2.2.2 One Dataset Indexed

Extending on a basic nested loop join approach, the indexed nested loop join [9] requires an index I_A for dataset A. The approach loops over dataset B and queries I_A for every object $b \in B$. Executing a query for each object is a substantial overhead, particularly if $B >> A$.

The seeded tree approach [21] also requires one dataset to be indexed with an R-Tree. The existing R-Tree I_A on dataset A is used to bootstrap building the R-Tree I_B on dataset B. After building the second index I_B, a synchronous traversal [7] is used for the join. Using the structure of I_A to build I_B ensures that the bounding boxes of both indexes are aligned, thereby reducing the number of bounding boxes that need to be compared. Improvements avoid memory thrashing [23] or use sampling to speed up building the R-Tree [20].

2.2.3 Unindexed

Disk-based approaches first partition both datasets and then join the resulting partitions in-memory. When assigning spatial objects to partitions, some objects may intersect with several partitions. Two different approaches, *multiple assignment* and *multiple matching*, have been developed to deal with this ambiguity.

Multiple Assignment: this strategy assigns each spatial object to all partitions it overlaps with (through duplication). The advantage is that the distance join only needs to compare objects inside one partition with each other (and

Terminology table	
highest level	level of the root node
lowest level	level of the leaf nodes
higher level	level closer to the highest level
lower level	level closer to the lowest level
A or dataset A or type A or blue	First spatial dataset
B or dataset B or type B or red	Second spatial dataset
T_A	R-Tree created for dataset A
L_{T_A}	Level of the tree T_A
MBR	Minimum Bounding Box (for any dimension)
node's self-mbr	mbr that contains the objects assigned to the node
node's light-mbr	contains the leaf level objects below the node
node's dark-mbr	contains objects assigned to the descendant nodes excluding leaf nodes below the node
nodes's mbr	is union of all the node's mbrs, i.e. light-, dark- and self-mbrs
$P^B_{T_A}$	the probability of assigning an object of dataset B to T_A

Table 1: Terminologies used in this paper.

not across partitions). Duplication, however, has major drawbacks, namely a) more comparisons need to be performed and b) because result pairs may be detected twice they need to be deduplicated either at the end (by keeping all results, thereby increasing the memory used - and deduplicating them at the end) or throughout the join [8].

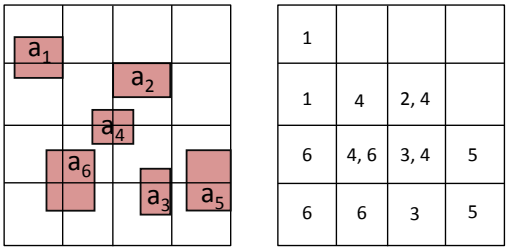


Figure 1: PBSM partitioning (left) and assignment (right)

PBSM [27] is the most recent and comparatively most efficient multiple assignment approach. As shown in Figure 1, PBSM partitions the entire space of both datasets into cells using a uniform grid. Every object from dataset A is assigned to all cells c_A it overlaps and all $b \in B$ are assigned to cells c_B respectively. After assigning all objects to cells, all pairs of cells c_A and c_B which have the same position are compared, i.e., all objects assigned to c_A and c_B are tested for intersection. Because objects are replicated intersections may be detected multiple times and hence deduplication needs to be performed.

The non-blocking parallel spatial join NBPS [22] algorithm produces the result tuples continuously as they are generated. NBPS distributes the tuples of the join relations to the data server nodes according to a spatial partitioning function. In contrast to PBSM, NBPS avoids duplicates so that the result can be returned immediately. More precisely, NBPS uses a revised reference point method to avoid the duplicates while TOUCH uses a hierarchical partitioning that not only avoids any replication (in comparison with NBPS) but also avoids the duplicates from the first stage of the distribution.

Multiple Matching: this strategy on the other hand assigns each spatial object only to one of the partitions it overlaps with. Hence when joining, objects in several partitions must be compared with each other, as an object at the border of one partition can potentially intersect with an object at the border of an adjacent partition.

The Scalable Sweeping-Based Spatial Join [5] is similar to PBSM but avoids replication. It partitions space into n equi-width strips in one dimension and maintains for every strip two sets LA_n and LB_n. It assigns each $a \in A$ that entirely fits into strip n to LA_n and for B and LB_n respectively. Finally, it uses an in-memory plane-sweep to find all intersecting pairs from LA_n and LB_n for all n.

An object o intersecting several strips will not be replicated but will instead be assigned to sets LA_{jk} and LB_{jk} respectively where j is the strip where o starts and k where o ends. When joining LA_n and LB_n all sets LA_{jk} and LB_{jk} with $j \leq n \leq k$ will also be considered in the plane-sweep.

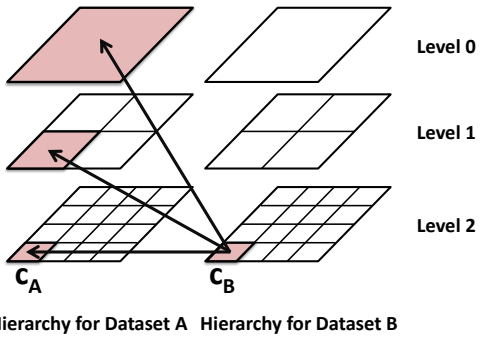


Figure 2: S3 space partitioning and multi-level join.

To avoid the replication of objects, S3 [17] maintains a hierarchy of L equi-width grids of increasing granularity as shown in Figure 2. In D dimensions the grid on a particular level l has $(2^l)^D$ grid cells and assigns each object of both datasets to a grid cell in the lowest level where it only overlaps one cell. To obtain this assignment, the algorithm starts with level L (Level 2 in figure 2) and moves up the levels until it finds the level where the object only overlaps one cell.

The algorithm maintains two hierarchies, H_A for dataset A and H_B for dataset B. Once all objects are assigned, the cells of H_A and H_B are joined. More precisely, a cell c_B of H_B is joined with its corresponding cell c_A of H_A and all the cells on higher levels of H_A enclosing c_A (example cells are shaded in Figure 2). Joining a cell with its counterpart as well as with cells on higher levels is repeated on all levels.

The process of joining the cells implies that the objects assigned to the highest level will be compared to all other objects (on all lower levels) and hence the more objects are assigned to the highest level, the more comparisons will be

needed. Datasets assigning more objects to levels closer to the leaf level will require fewer comparisons for the join.

3. CHALLENGE

The development of TOUCH is driven by the requirements of the computational neuroscientists we collaborate with. In order to better understand how the brain works, the neuroscientists build biophysically realistic models of a neocortical column on the molecular level and simulate them on a BlueGene/P with 16K CPUs. Each of the models contains several thousand neurons where each neuron and its branches are modeled as thousands of cylinders. Figure 3 shows a cell morphology, with cylinders modeling the dendrite and axon branches in three dimensions.

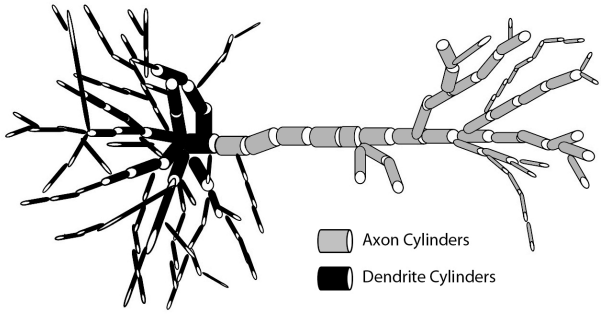


Figure 3: Neuron modeled with cylinders.

The models are built based on the analysis of real rat brain tissue. The structure of the neuron is rebuilt using brightfield microscopy and their electrophysiological properties are obtained by the patch clamp technique [14]. The locations of synapses (i.e., points where impulses leap over between neurons), however, are not known, cannot be determined by the above techniques and thus need to be added in a post-processing step. Placing the synapses is a one-off operation executed only once for each model built. The synapse locations are determined by the following rule: a synapse is placed wherever a neuron's dendrite is within a certain distance of another neuron's axon. Previous neuroscience research has confirmed that a realistic model of the brain is built by following this rule [18].

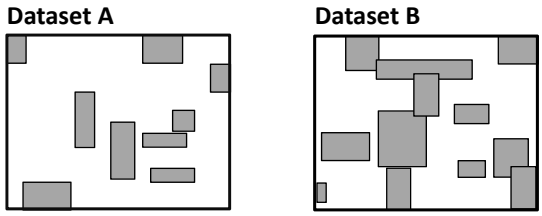
The problem of placing synapses therefore translates to a spatial distance join between two unindexed and unsorted datasets, one dataset containing cylinders representing axons and one containing cylinders representing dendrites. The distance join is only executed once on each new model and thus no data structures can be shared between distance joins on different models. Because the 16K cores of the BlueGene/P cannot access the disk concurrently, the join has to be performed in memory alone. Furthermore, this independency is advantageous for data-parallel algorithms that can be executed on GPUs [25]. To do so, and because this is an embarrassingly parallel problem, the dataset is split into 16K contiguous subsets, each subset is loaded in the memory of a core and the distance join is performed locally (independent of the other cores and thus massively parallel).

The models currently built and simulated by the neuroscientists contain at most few million neurons, but the ultimate goal is to simulate the human brain with approximately 10^{11} neurons. To achieve this goal, the number of neurons needs to increase until the model has the same neuron and synapse density as the human brain. The higher density of a dataset modeling the human brain, as opposed to those modeling the rat brain, will substantially increase the selectivity of the distance join between the two datasets, as more synapses are found in the former than in the latter. An efficient distance join approach is therefore pivotal already today and the advancement of neuroscience research requires an in-memory spatial join that will scale to the increasingly high selectivity of arising models of the brain/neuroscience datasets.

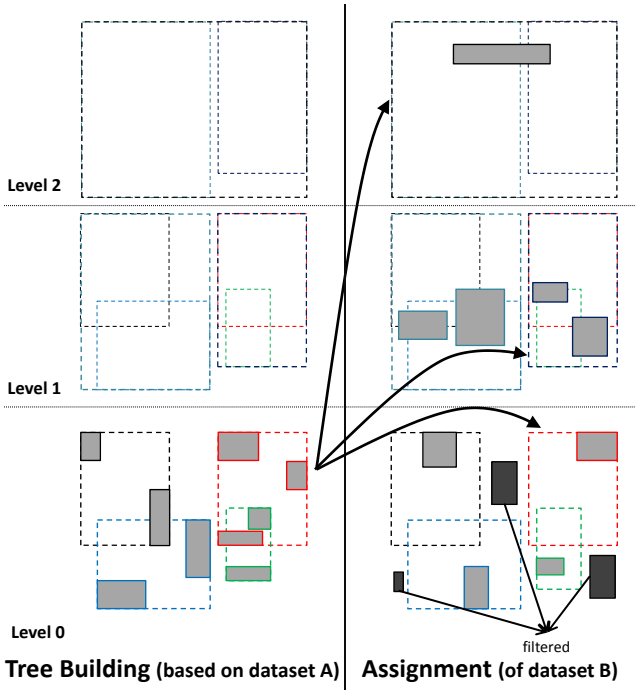
The particular application notwithstanding, an efficient in-memory spatial join is important for many applications: after all, every disk-based spatial join needs to perform an in-memory join.

4. TOUCH

Given the lack of adequate in-memory spatial join approaches we develop TOUCH, a new in-memory spatial join algorithm for two unsorted and unindexed spatial datasets, DS_1 and DS_2. We translate the distance join required by our motivating example into a spatial join that detects intersection between objects: instead of finding all pairs of objects $d_1 \in DS_1$ and $d_2 \in DS_2$ such that $distance(d_1, d_2) \leq \epsilon$, we increase the size of all objects of one dataset, say DS_1, by ϵ and test for intersection with objects of DS_2 [15].



(a) The datasets A and B



(b) Tree building, assignment and joining phases

Figure 4: The three phases of TOUCH: building the tree, assignment and joining.

It is established practice to perform a spatial join in two phases: filtering followed by refinement [26]. Like most other approaches, TOUCH focuses on the filtering phase in which all objects are approximated by bounding boxes. Our solution can be combined with any off-the-shelf solution to the second refinement phase, which takes into account the exact object shapes (e.g., cylinders, spheres, etc.).

4.1 EVERGREEN Ideas

Big data processing calls for exploiting the ubiquitous parallel processors, e.g. GPU and CPU. These architectures are well designed for independent balanced workloads. Thanks to the TOUCH algorithm, the workloads are independent. However, we need balanced workloads as well. The reTOUCH algorithms are designed with the idea of creating balanced workload by reducing the asymmetry of the TOUCH algorithms as much as possible. The algorithms are explained in section Section 5 in detail. The followings are the TOUCH ideas:

In developing TOUCH, we are inspired from previous disk-based approaches, which can also be used in main memory. However, we aim to combine the benefits and avoid the pitfalls of previous approaches. In particular, we want to avoid multiple assignment (as in PBSM), because it replicates objects and therefore (a) increases the memory footprint; (b) requires multiple comparisons; and (c) makes it necessary to deduplicate the results.

At the same time we want to reduce the number of comparisons of multiple matching approaches and we therefore want to use data-oriented partitioning instead of space-oriented partitioning as S3 does.

To further reduce the number of comparisons, we also use filtering, a concept used by S3. In S3, objects from the second dataset B are discarded if they intersect only with cells that contain no objects from dataset A. If object $b \in B$ is only overlapping cells that contain no object from A then b cannot possibly intersect with any object from A. Hence b does not need to be considered further.

The main innovation of TOUCH lies in the fact that it directly assigns objects of the second data set to the data-oriented index of the first. By doing so it avoids the problems caused by excessive index-overlap in other data-oriented approaches (R-Tree) as well as the problems of space-oriented indexing approaches (like S3). As we will explain, the combination of data-oriented partitioning during index-building on the first dataset with hierarchical assignment of the second dataset leads to significantly fewer comparisons and speeds up the join itself.

The fact that we design our approach for use in memory gives us more degrees of freedom. We no longer have to align the data structures for the disk page size, but can choose the size of the data structures used more flexibly (partitions of arbitrary size, variable fanout, etc.).

4.2 Proof of Correctness

We may want to have a similar subsection for reTOUCH algorithms

5. EVERGREEN ALGORITHMS

In this section, we are devising our novel algorithms, called dTOUCH, cTOUCH, reTOUCH and ultimately re*TOUCH algorithm. Since we are dealing with big data, we need to exploit the ubiquitous parallel processors, e.g. GPU and CPU. These architectures are well designed for independent balanced workloads. Thanks to the TOUCH algorithm, the workloads are independent. However, we need balanced workloads as well. The EVERGREEN algorithms are designed with the idea of creating balanced workload by reducing the asymmetry of the TOUCH algorithms as much as possible. These algorithms create indexes that produce even workloads for the actual job. Each of the following designs has pros and cons that worth studying.

	Grid at Descendant	Grid at Ancestor
Top-Down	Case 1	Case 3
Bottom-Up	Case 4	Case 2

Table 2: Tree traversal and grid placement of SGH join.

In order to explain the EVERGREEN algorithms we need to have an overview of the TOUCH algorithm. Therefore, next subsection briefly explains the TOUCH algorithm and the subsequent subsections explain our novel algorithms. For clarity, we assume that the lowest level of a tree is the leaf level and the highest level is the level of root node and accordingly lower and higher levels can be defined. Please find table 1 the list of terminologies used in this paper.

5.1 Detail information

We have light, dark, self-dark and black mbrs. black mbr is union of all the other mbrs. light mbr contains the leaf level objects. dark mbr contain assigned object in my descendants and not leaf. self-dark only assigned objects to the node. light mbr is used for faster shrinking after assignment of all the objects of a type of a leaf node. black mbr for the faster assignment. self-dark and dark for those objects that cannot be filtered because they had some overlap on the ancestors assigned objects.

SGH: case 1 and 4.

- Case 1: Each node joining with descendants' grids on demand.
- Case 2: Each node joining with ancestors' grids on demand.
- Case 3: Each node lets descendants join with its own grid.
- Case 4: Each node lets ancestors join with its own grid.

Grid size: every node has an MBR that contains all the objects assigned to the node. For each node we divide each dimension of its MBR to $\sqrt[D]{\frac{|MBR|}{|AVE|}}$ partitions, such that D is number of dimensions, $|MBR|$ is the size of the MBR and $|AVE|$ is the average size of the objects inside the node.

5.2 TOUCH algorithm

The TOUCH algorithm consists of three steps: Tree construction (Algorithm 1), Assignment (Algorithm 2) and Joining (Algorithm 3) steps. The algorithms 1, 2 and 3 describes the TOUCH algorithm in a way that smooths our EVERGREEN algorithms explantions that follows subsequently.

Algorithm 1 builds an R-Tree from the objects of $dataset_A$, i.e. T_A. This algorithm first partitions the objects by using Hilbert Sort. Then constuct a tree from the created partitions in which each node contains an MBR that covers all the objects in its descendant nodes.

Algorithm 2 assigns the objects of $dataset_B$ to the internal nodes of the constructed tree of the algorithm 1, T_A. The

Algorithm 1 TOUCH algorithm, building T_A

```
1: function TreeConstruction(dataset)                    ▷ dataset_A
2:     nextInput = (dataset, level = 0)
3:     while nextInput is not empty do
4:         (ds, level) = nextInput
5:         SpatialSort(ds)                               ▷ e.g. Hilber Sort
6:         if level = 0 then
7:             split ds into partitions of size leafSize.
8:         else
9:             split ds into fanout number of partitions.
10:        for all part ∈ partitions do
11:            MBR = calculate the mininum bounding rectangle that covers all the objects inside part.
12:            tree ← [part, level]
13:            nextInput ← (MBR, level = level + 1)
```

Algorithm 2 TOUCH algorithm, Assignment step

```
1: function assignment(dataset_B)
2:     for all b in dataset_B do
3:         currentNode = root node of T_A
4:         S = set of nodes ∈ children of currentNode that b intersects
5:         if |S| = 0 then                                ▷ b is filtered
6:             continue
7:         if |S| = 1 then                    ▷ go one level lower in T_A
8:             currentNode = S
9:             if currentNode is a leaf node then
10:                assign b to currentNode and continue
11:            else
12:                GOTO line 4
13:        if |S| > 1 then                                ▷ assign to this level
14:            assign b to currentNode and continue
```

goal of algorithm 2 is to do a *complete assignment* (defined in theorem 1) of the objects to the lowest internal node of T_A.

Theorem 1. *An assignment of an object o to an internal node n is complete iff among all the siblings of node*

Proof. The assignment is complete because neither the siblings nor the ancestors of node n has an object that overlaps with o □

Algorithm 3 TOUCH algorithm, Join step

```
1: function TJoin(treeNode)                              ▷ root node of T_A
2:     O_B = objects of dataset_B assigned to the treeNode
3:     MBR = MBR of treeNode
4:     gridHash = Divide the MBR of treeNode to grid cells with the given resolution
5:     for all objects O_A of dataset_A ∈ descendant leaf nodes of treeNode do
6:         SGH(O_A, gridHash)                            ▷ Spatial Grid Hash join
7:     for all childNode ∈ treeNode do
8:         TJoin(childNode)
```

Algorithm 3 performs the required pairwise comparisons according to the assignment of the objects of $dataset_B$ to the T_A. The algorithm employs the Spatial Grid Hash join (SGH) for joining the objects of $dataset_B$ in each internal node of T_A with the objects of $dataset_A$ in its (the internal node) descendant leaf nodes. The algorithm starts by traversing T_A top-down. When visiting a node, to perform SGH, the algorithm constructs the grid for the visited node and the objects in its descendants probe the grid. Succinctly, each node lets descendants join with its own grid.

5.3 dTOUCH algorithm (double TOUCH)

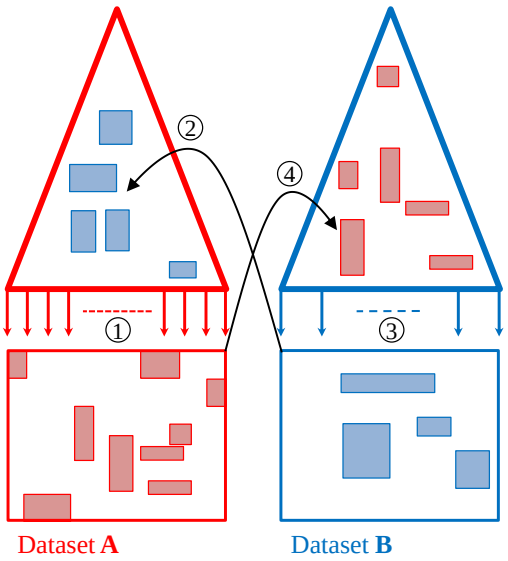


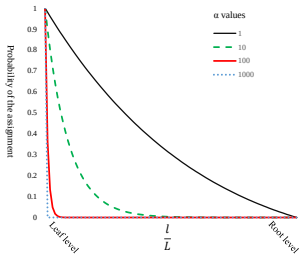
Figure 5: dTOUCH algorithm procedure

This modification is based on building two separate R-trees. First step is building R-tree using objects from dataset A, the same as in TOUCH. Then, we assign objects of

dataset B to this tree. During the assignment phase the level of the node where we decide to assign our object represents the workload for the joining step. Since every assigned object must be checked with all the objects in the descendant leafs nodes, the lower level assignment requires less number of comparisons for joining[1]. In dTOUCH we restrict assignment to high levels of the tree. The restriction avoids assigning many objects of dataset B to higher levels of the tree built from objects of dataset A, T_A. And in return, for the objects of dataset B that has not been assigned to T_A, we have to create another tree, T_B, to compensate the missing results of the not assigned objects of the dataset B. The decision of whether assigning an object of B to level l of the T_A that has L_{T_A} levels is based on the following probability, $P^B_{T_A}$:

$$P^B_{T_A} = e^{-\alpha \times \frac{l}{L_{T_A}}} \times (1 - \frac{l}{L_{T_A}})$$

$P^B_{T_A}$ is the probability of assigning an object of dataset B to the tree of dataset A, T_A. The equation consists of an exponent part and a coefficient to the exponent part. The former part of the equation creates a fast decreasing curve starting from one and the latter part creates an end (when we are at the root level) of zero (to avoid assignment to the root) for the curve.



assignment coefficient α.

This figure illustrates how different values of the coefficient α influences the assignment. In order to avoid many assignment to the root node ($\alpha = 1$) on one hand and also let some objects to be assigned to high levels ($\alpha \geq 100$) on the other hand we choose $\alpha = 10$.

After assigning objects of dataset B to T_A, dTOUCH creates another R-Tree, T_B, for the remaining, not previously assigned, objects of dataset B. Then, dTOUCH similarly assigns all the objects of dataset A to T_B. Finally, dTOUCH joins each tree, T_A and T_B, with their assigned objects similar to the algorithm 3.

Algorithm 4 dTOUCH algorithm, Assignment restriction part

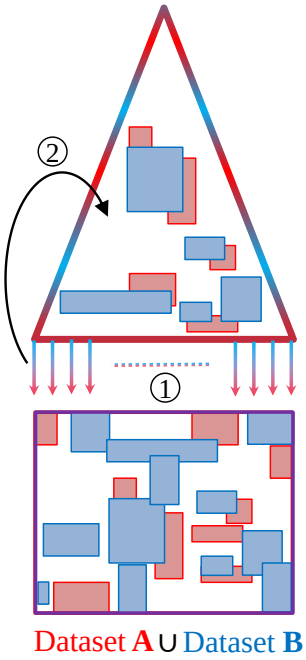
```
1:  function ASSIGNMENT(dataset_B)
13:     if |S| > 1 then ▷ assign with the following condition
14:         r = generate a uniform random number ∈ [0, 1]
15:         l = level of currentNode
16:         if r ≤ P^B_{T_A} then
17:             assign b to currentNode
18:             delete b from dataset_B
19:         continue
```

Figure 5 illustrates the steps of dTOUCH algorithm: 1) constructing T_A; 2) Assigning the objects of dataset B to the internal nodes of T_A following algorithm 4; 3) constructing R-Tree for the remaining objects B' of dataset B, $T_{B'}$; 4) assigning the objects of dataset A to $T_{B'}$ following algorithm 2.

[1]The number of descendants grows exponentially with the level.

5.4 cTOUCH algorithm (complex TOUCH)

In our second attempt to reduce the asymmetric exist in TOUCH instead creating two separate trees we create a single tree by allowing the objects of both datasets to be assigned to internal nodes of the tree. Therefore, cTOUCH algorithm constructs one symmetric R-tree, using both datasets simultaneously.



To explain cTOUCH we use colors in our definitions to distinguish datasets and MBRs types. We assume objects of dataset A and B be red o^r and blue o^b respectively. Following this coloring we will define six colors of MBRs. Given these assumptions we construct a single tree that contains both datasets objects assigned to any level of the tree by performing the tree construction (first step of TOUCH algorithm) on a dataset that is a merge of both datasets A and B (Early merging).

This figure illustrates the steps of cTOUCH algorithm: 1) constructing R-Tree for dataset $A \cup B$, $T_{A \cup B}$; 2) Moving the objects of the leaf level to the internal levels of $T_{A \cup B}$ following algorithm **??**.

5.5 Alvis cTOUCH

The key idea of rewriting TOUCH code is making the algorithm symmetric to A and B datasets. In this implementation we will try to build one symmetric R-tree, using both datasets as equally as possible. For pretty illustrating the principals of the approach lets introduce some colors and definitions. Let objects (o) from dataset A be red objects (o) and from dataset B - blue objects (o). Each objects has its own MBR ($\square$) that has corresponding color. Let MBR of the node i on the level l be red ($v^l_i \leftarrow \square^l_i$) if it was built as intersection of all red MBRs of the children ($\square^l_i = \bigcup \square^l_{i-1}$). Lets MBR of the node be blue if it was built from blue ones. Let MBR be black ($\square^l_i$) if it was build as intersection of blue and red MBRs. Let the assigned red objects be dark red ($\blacksquare^l_i$) and assigned blue - dark blue ($\blacksquare^l_i$).

5.5.1 Overview

Finished with colors, now the introduction to the algorithm goes. The algorithm consists of the same steps as original TOUCH: building, assignment and joining steps. During the building step we take all objects of both colors, build leaf nodes (create initial groups of objects) and build one mutual R-tree using Hilbert curve for indexing starting from leafs and creating two MBR per each node. Assignment phase is about taking all the objects from the leafs of

the tree and assigning them to the tree traversing from the root to the leafs. At the joining step for each node we check the intersection of assigned object with all assigned objects which were assigned to the descenders of the node.

5.5.1.1 *Reminder.*

$\square_i^l$ - MBR where l - level in the tree, i - index of the node where it must be located.

5.5.2 *Building phase*

See the algorithm 5.

Algorithm 5 cTOUCH algorithm, Building phase

$\square_i^0 \leftarrow o_i$
$\square_i^0 \leftarrow o_i$
$\forall j \quad v_j^0 \leftarrow \square_i^0, \square_i^0 \quad i \in \{jC, jC+1, ..., (j+1)C\}$ ▷ C - leaf capacity
$\forall v_j^0 \quad \square_j^0 \leftarrow \square_j^0 \cup \square_j^0$
$\Omega \leftarrow v_j^0 \quad \forall j$
while size$(\Omega) \neq 1$ **do**
    $v_i^l \leftarrow \Omega$
    Sort v_i^l by $\square_i^l$ using Hilbert curve
    $\forall n \quad \omega_n \leftarrow v_i \quad i \in \{jF, j(F+1), ..., (j+1)F\}$ ▷ F - fanout
    for all j **do**
        for all $\square_i^l, \square_i^l \leftarrow \omega_n$ **do**
            $\square_j^{(l+1)} \leftarrow \bigcup_i \square_i^l$
            $\square_j^{(l+1)} \leftarrow \bigcup_i \square_i^l$
            $v_j^{(l+1)} \leftarrow \square_j^{(l+1)}, \square_j^{(l+1)}$
            $E \leftarrow (i, j)$ ▷ E - set of edges of the tree
            $\Omega \leftarrow v_j^{(l+1)}$
Root $\leftarrow \Omega$

5.5.3 *Assignment phase*

See the algorithm 6.

5.5.4 *Joining phase*

See the algorithm 7.

5.6 reTOUCH algorithm (redoing TOUCH)

reTOUCH algorithm avoids the so-called early merging of cTOUCH while creates a single tree for both datasets that contains their objects in any level. reTOUCH algorithm first constructs a tree that is formed[2] by the objects of dataset A, then assigns the objects of dataset B to this tree without any level restriction. Given this tree that contains only objects of dataset B we reform the tree, then assign the objects of dataset A.

The last assignment, of dataset A, has a further filtering feature that we exploit. While reTOUCH assigns each object a of A by traversing the T_B starting from the root node. At each level, we move one level down iff a overlaps with a single node's mbr. However, in the top-down traversal we may reach to a level that a has no overlap with any node's mbr while a had overlap with a node's self mbr before. Therefore, a must be assigned to the lowest level that it overlaps with a node's self mbr of that level. Since a has no overlap with any object below that level, we can assign a in a way that avoids the redundant future comparisons of a to the lower levels. Therefore, in reTOUCH we distinguish

[2]without assigning them to the tree

Algorithm 6 cTOUCH algorithm, Assignment phase

for all v_i^0 **do**
    for all $o \leftarrow v_i^0$ **do**
        v_j^l = Root
        loop
            $\omega = \{v_k^{l-1} : o^r \cap (\square_k^{(l-1)} \cup \blacksquare_k^{(l-1)}) \neq \varnothing, (k, j) \in E\}$
            if $\omega = \varnothing$ **then**
                Filter o
                Break
            if size$(\omega) = 1$ **then**
                $v_j^l \leftarrow \omega$
            if size$(\omega) > 1$ or v_j^l - leafnode **then**
                $v_j^l \leftarrow o$
                for $l' \geq l$ **do**
                    $\blacksquare_k^{l'} \leftarrow \square_j^l \quad \forall (k, j) \in E$
                Break
    Delete $\square_i^0$
    for all $l > 0$ and $k : (k, i) \in E$ **do**
        $\square_k^l \leftarrow \bigcup_j \square_j^{(l-1)} \quad \forall j : (j, k) \in E$
    for all $o \leftarrow v_i^0$ **do**
        v_j^l = Root
        loop
            $\omega = \{v_k^{l-1} : o \cap (\square_k^{(l-1)} \cup \blacksquare_k^{(l-1)}) \neq \varnothing, (k, j) \in E\}$
            if $\omega = \varnothing$ **then**
                Filter o
                Break
            if size$(\omega) = 1$ **then**
                $v_j^l \leftarrow \omega$
            if size$(\omega) > 1$ or v_j^l - leafnode **then**
                $v_j^l \leftarrow o$
                for $l' \geq l$ **do**
                    $\blacksquare_k^{l'} \leftarrow \square_j^l \quad \forall (k, j) \in E$
                Break
    for all $l > 0$ and $k : (k, i) \in E$ **do**
        $\square_k^l \leftarrow \bigcup_j \square_j^{(l-1)} \quad \forall j : (j, k) \in E$

Algorithm 7 cTOUCH algorithm, Joining phase

for all v_i^l **do**
    for all $o \leftarrow v_i^l$ **do**
        for all $v_j^{l'} : l' \leq l, j \in E$ **do**
            for all $o \leftarrow v_j^{l'}$ **do**
                if $o \cap o \neq \varnothing$ **then**
                    Result $\leftarrow (o, o)$
    for all $o \leftarrow v_i^l$ **do**
        for all $v_j^{l'} : l' \leq l, j \in E$ **do**
            for all $o \leftarrow v_j^{l'}$ **do**
                if $o \cap o \neq \varnothing$ **then**
                    Result $\leftarrow (o, o)$
    for all $o, o \leftarrow v_i^l$ **do**
        if $o \cap o \neq \varnothing$ **then**
            Result $\leftarrow (o, o)$

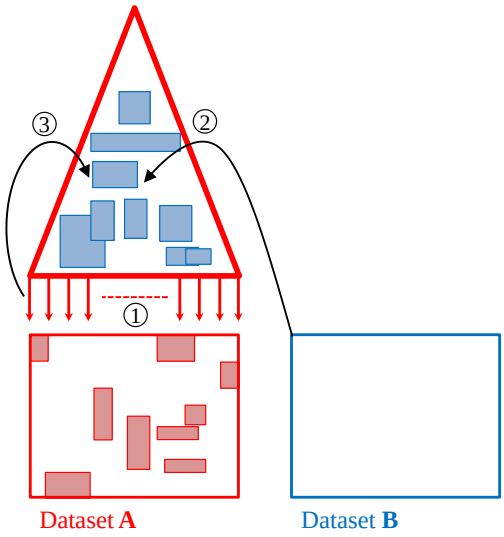


Figure 6: reTOUCH algorithm procedure

these kind of assignments from the *universal* assignments that we did for all the other algorithms. As a result of this further filtering feature we introduce two types of assignment: *universal* and *top* assignment. The latter assignment indicates that there is no overlap with objects below, in contrast to the former assignment that has this below overlap.

Figure 6 illustrates the steps of reTOUCH algorithm: 1) constructing T_A; 2) assigning objects of dataset B to T_A following algorithm 2; 3) Moving the objects of the leaf level to the internal levels of T_A following algorithm **??**.

5.7 Balancing control

In this subsection we explain how we enforce balancing by defining a cost function per object.

6. IMPLEMENTATION

In order to implement TOUCH we build in a first phase a tree based on dataset A. Each node in the tree contains a triple {*children*, *MBR*, *entities*}. The leaf nodes contain as entities the objects of dataset A and each node's MBR encloses all objects in it. The MBR of an inner node is computed recursively based on the MBRs of its children.

In the second phase, the objects $b \in B$ are distributed and stored as the entities of the inner nodes of the tree. Following Algorithm **??**, each object b is assigned to the node with the smallest MBR that still covers it.

6.1 Partitioning

In the first phase TOUCH partitions the objects of dataset A into buckets and builds a tree, similar to an R-Tree [13].

In TOUCH we use the Sort-Tile-Recursive approach [19] to group the objects. This technique first sorts the MBRs on the first dimension. Then STR divides the dataset into $P^{1/D}$ slabs, where P is the number of partitions and D is the number of dimensions, according to their current order. Finally, STR recursively processes the remaining slabs using the remaining $D-1$ dimensions. STR typically produces leaf nodes with the smallest MBRs [19] (also on higher levels) and thus allows for more effective filtering.

6.2 Design Parameters

In the course of building the data structures, i.e., the tree, and ultimately joining the datasets several parameters can be set and tuned. In the following we discuss what they are and how they can have an impact on the performance.

6.2.1 *Tree Parameters*

The tree built on dataset A in the first phase of the join resembles an R-Tree and hence has tunable parameters like the size of the leaf nodes and the fanout, i.e., the number of children each node has (except the leaf nodes). The size of the inner nodes depends on the assignment of dataset B and can hence not be known a priori.

For the disk-based R-Tree, the fanout defines the node size and is chosen such that the nodes fit on an integer number of disk pages. Because TOUCH is developed for memory, disk restrictions no longer apply and we can choose fanout and node size more freely. The choice of the fanout, however, has an impact on the performance.

6.2.2 *Local Join Parameters*

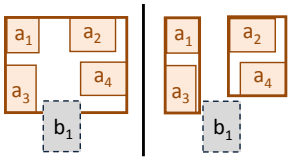


Figure 7: Filtering.

Local join is performed using Spatial Grid Hash (SGH) with dynamic resolution. Lets assume that SGH resolution is calculated for the node V with assigned objects $a_1, a_2, ..., a_m$.

$MBR(V) = \cup_{i=1..m} MBR(a_i)$

$MBR(V)$ is the universe for SGH in V. The best resolution is one with each cell having equal size with an average assigned object, since in that case SGH does not have many duplications and it does not have many objects assigned to the single cell. For that reason we calculate the average size of assigned objects $a_1, ..., a_m$ for each dimension and use that number for calculation of the resolution:

$$MBR_x(V) = \frac{1}{m} \sum_{i=1..m} MBR_x(a_i)$$

$$MBR_y(V) = \frac{\sum_{i=1..m} MBR_y(a_i)}{m}$$

$$MBR_z(V) = \frac{\sum_{i=1..m} MBR_z(a_i)}{m}$$

6.2.3 *Join Order*

Given two datasets A and B, deciding which dataset to use first to build the tree and which one to assign to the inner nodes of the tree is pivotal for performance.

To improve filtering, the decision would ideally be based on the density of both datasets: the sparser the first dataset, the more objects of the second dataset may be filtered. Not knowing the density of the datasets a priori, we can still make a reasonable assumption and take the smaller dataset as the first dataset. If it has the same (or a bigger) spatial extent than the bigger dataset, then it will be sparser and allow for more filtering. If its spatial extent is smaller than the one of the bigger dataset, then the difference in extent allows for effective filtering as well.

Using the smaller dataset first will also help to speed up building the hierarchical structure as well as reduce its size in memory.

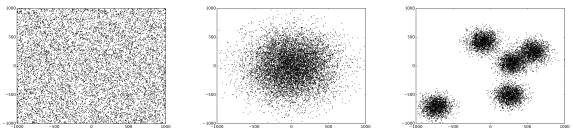
c 

Figure 8: Types of distributions of objects

7. EXPERIMENTAL EVALUATION

7.1 Setup

7.2 Data description

We used synthetic data for our experiments with different distributions of objects and. For each size of a dataset and for each distribution type we generated 10 samples. Three distributions are used: uniform , gauss and clustered (pic. 8). Clustered distribution is objects that are randomly assigned to one of the cluster centers in the space and have a gauss distribution around the center of the cluster.

7.3 Experimental Methodology

8. CONCLUSIONS

In this paper we identify the lack of efficient approaches for in-memory spatial joins. We demonstrate that the two in-memory approaches, namely the nested-loop and the plane-sweep, are inefficient and that disk-based approaches are not efficient either when applied in main memory. An exception is PBSM, which is fast but needs excessive memory.

We develop TOUCH, a spatial join algorithm that combines the advantages while avoiding the pitfalls of previous approaches. TOUCH avoids a space-oriented partitioning on the large scale and uses a data-oriented partitioning to organize both datasets. To avoid object replication, objects of one dataset are used to construct an R-Tree-like hierarchy, while those of the other are assigned to the level where they are fully contained by an MBR. The partitions on different levels are joined using a space-oriented partitioning.

We experimentally demonstrate that our approach is faster and has a substantially smaller memory footprint than the previous state of the art with real and synthetic datasets. Our results also indicate that TOUCH is scalable to larger and denser datasets, a key issue in scaling up the neuroscience application we are motivated from. Thus, TOUCH is recommendable as a method of choice for such applications. Furthermore, as we only make a few assumptions about dataset characteristics, with TOUCH we provide a general-purpose solution applicable on any spatial dataset.

9. REFERENCES

[1] W. G. Aref and H. Samet. A Cost Model for Query Optimization Using R-Trees. In *GIS '94*.

[2] W. G. Aref and H. Samet. Cascaded Spatial Join Algorithms with Spatially Sorted Output. In *GIS '96*.

[3] W. G. Aref and H. Samet. Hashing by Proximity to Process Duplicates in Spatial Databases. In *CIKM '94*.

[4] L. Arge, M. de Berg, H. J. Haverkort, and K. Yi. The Priority R-tree: a practically efficient and worst-case optimal R-tree. In *SIGMOD '04*.

[5] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable Sweeping-Based Spatial Join. In *VLDB '98*.

[6] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: an efficient and robust access method for points and rectangles. *SIGMOD Record*, 19(2):322–331, 1990.

[7] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient Processing of Spatial Joins Using R-Trees. In *SIGMOD '93*.

[8] J.-P. Dittrich and B. Seeger. Data Redundancy and Duplicate Detection in Spatial Join Processing. In *ICDE 2000*.

[9] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison Wesley, 3rd edition, 2000.

[10] A. Farris, A. Sharma, C. Niedermayr, D. Brat, D. Foran, F. Wang, J. Saltz, J. Kong, L. Cooper, T. Oh, T. Kurc, T. Pan, and W. Chen. A Data Model and Database for High-resolution Pathology Analytical Image Informatics. *Journal of Pathology Informatics*, 2(1):32, 2011.

[11] Y. J. García, M. A. López, and S. T. Leutenegger. A Greedy Algorithm for Bulk Loading R-trees. In *GIS '96*.

[12] S. Gnanakaran, H. Nymeyer, J. Portman, K. Y. Sanbonmatsu, and A. E. Garcia. Peptide folding simulations. *Current Opinion in Structural Biology*, 13(2):168–174, 2003.

[13] A. Guttman. R-trees: a Dynamic Index Structure for Spatial Searching. In *SIGMOD '84*.

[14] O. P. Hamill, A. Marty, E. Neher, B. Sakmann, and F. J. Sigworth. Improved Patch-clamp Techniques for High-resolution Current Recording from Cells and Cell-free Membrane Patches. *Pflügers Archiv European Journal of Physiology*, 391:85–100, 1981.

[15] E. H. Jacox and H. Samet. Spatial Join Techniques. *ACM TODS '07*.

[16] I. Kamel and C. Faloutsos. Hilbert R-tree: An Improved R-tree using Fractals. In *VLDB '94*.

[17] N. Koudas and K. C. Sevcik. Size Separation Spatial Join. In *SIGMOD '97*.

[18] J. Kozloski, K. Sfyrakis, S. Hill, F. Schürmann, C. Peck, and H. Markram. Identifying, Tabulating, and Analyzing Contacts Between Branched Neuron Morphologies. *IBM Journal of Research and Development*, 52(1/2):43–55, 2008.

[19] S. Leutenegger, M. Lopez, and J. Edgington. STR: a Simple and Efficient Algorithm for R-Tree Packing. In *ICDE '97*.

[20] M.-L. Lo and C. V. Ravishankar. Spatial Hash-Joins. In *SIGMOD '96*.

[21] M.-L. Lo and C. V. Ravishankar. Spatial Joins Using Seeded Trees. In *SIGMOD '94*.

[22] G. Luo, J. F. Naughton, and C. J. Ellmann. A non-blocking parallel spatial join algorithm. In *ICDE*, pages 697–705, 2002.

[23] N. Mamoulis and D. Papadias. Slot Index Spatial Join. *IEEE TKDE*, 15(1):211–231, 2003.

[24] P. Mishra and M. H. Eich. Join Processing in Relational Databases. *ACM Computing Surveys*, 24(1):63–113, 1992.

[25] S. Nobari, T.-T. Cao, P. Karras, and S. Bressan. Scalable parallel minimum spanning forest computation. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of*

Parallel Programming, PPoPP '12, pages 205–214, New York, NY, USA, 2012. ACM.

[26] J. Orenstein. A Comparison of Spatial Query Processing Techniques for Native and Parameter Spaces. In *SIGMOD '90*.

[27] J. M. Patel and D. J. DeWitt. Partition Based Spatial-Merge Join. In *SIGMOD '96*.

[28] F. Preparata and M. Shamos. *Computational Geometry: An Introduction*. Springer, 1993.

[29] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *VLDB '87*.

[30] M. Ubell. The Montage Extensible DataBlade Architecture. In *SIGMOD '94*.