GIPSY: Joining Spatial Datasets with Contrasting Density

Mirjana Pavlovic[†], Farhan Tauheed[†‡], Thomas Heinis[†], Anastasia Ailamaki[†]

[†]*Data-Intensive Applications and Systems Lab, École Polytechnique Fédérale de Lausanne, Switzerland*

[‡]*Brain Mind Institute, École Polytechnique Fédérale de Lausanne, Switzerland*

ABSTRACT

Many scientific and geographical applications rely on the efficient execution of spatial joins. Past research has produced several efficient spatial join approaches and while each of them can join two datasets, the problem of efficiently joining two datasets with contrasting density, i.e., with the same spatial extent but with a wildly different number of spatial elements, has so far been overlooked. State-of-the-art data-oriented spatial join approaches (e.g., based on the R-Tree) suffer from degraded performance due to overlap, whereas space-oriented approaches excessively read data from disk.

In this paper we develop GIPSY, a novel approach for the spatial join of two datasets with contrasting density. GIPSY uses fine-grained data-oriented partitioning and thus only retrieves the data needed for the join. At the same time it avoids the overlap related problems associated with data-oriented partitioning by using a crawling approach, i.e., without using a hierarchical tree. Our experiments show that GIPSY outperforms state-of-the-art disk-based spatial join algorithms by a factor of 2 to 18 and is particularly efficient when joining a dense dataset with several sparse datasets.

1. INTRODUCTION

An increasing number of scientific or GIS applications depend on the efficient execution of spatial join operations. In geographical applications, for example, spatial joins are executed to determine the intersection or proximity between geographical features [26], i.e., landmarks, roads, etc. Medical imaging applications need an efficient spatial join to determine the proximity between cancerous cells [8] and in neuroscience the join is performed to find the intersection of neuron branches [21].

Many efficient approaches for disk-based spatial joins [5, 23] have been developed in the past. Unfortunately none of them can efficiently and scalably join two spatial datasets of substantially different density, i.e., of similar spatial extent but with a vastly different number of spatial elements. Doing so, however, is a crucial operation for several applications: it is needed to efficiently add a small number of roads or few elements to GIS datasets, to add the branches of one neuron to a spatial model of the neocortex and many other applications. The efficiency of the join is pivotal as it is oftentimes executed repeatedly to join several sparse datasets with one dense dataset.

To define the problem more formally, our goal is to develop an approach for repeated spatial joins of sparse datasets with one dense dataset. Given several sparse datasets A_i and a dense dataset B where $A_i << B$ (i.e., their spatial extent is similar, but the number of elements differ), the approach finds all pairs of spatial elements $a_k \in A_i$ and $b \in B$ so that a_k and b intersect. While any previously developed method [12] can be used to join a dataset A_i (with few elements) and B (with a massive number of elements), the state of the art is inefficient, as we will show with motivating experiments.

With the sparse datasets A_i repeatedly joined with the dense dataset B, building an index on B or on all A_i and B will speed up the join operation. The fundamental problem of existing approaches, however, is that with a very small A_i, only a small subset of B needs to be retrieved (and tested against A_i). Existing approaches based on space-oriented partitioning (e.g., PBSM [23]) create coarse-grained partitions and consequently the entire dataset B needs to be read for a join, leading to excessive disk accesses. Approaches based on data-oriented partitioning allow for a more fine-grained partitioning of the data, but require hierarchical trees (e.g., the synchronized R-Tree [5], indexed nested loop on the R-Tree [7]) to access the data and thus suffer from well documented problems like overlap and dead space, also resulting in excessive disk accesses.

We propose GIPSY, a novel approach that uses fine-grained data-oriented partitioning and thereby enables the join to read from B only the small subset needed. It avoids the overlap inherent in data-oriented partitioning by using an efficient crawling technique [22, 25] which is also used for range queries on spatial data. With this novel combination of crawling with data-oriented partitioning, GIPSY achieves a 2 to 18× speedup compared to the fastest approaches like the indexed nested loop [7] and PBSM [23] when joining several A_i with B.

The remainder of this paper is structured as follows. We first discuss related work in Section 2 and then motivate our work with an example application from neuroscience in Section 3. With an initial set of measurements we also demonstrate the shortcomings of the state of the art. In Section 4 we then explain our approach, GIPSY, and evaluate it extensively in Section 5. We conclude in Section 6.

SSDBM '13, July 29 - 31 2013, Baltimore, MD, USA

2. RELATED WORK

Several spatial join approaches have been developed for disk [12]. In the following, we discuss related work and categorize it according to its use of data- or space-oriented indexes.

2.1 Data-oriented Partitioning

Spatial join methods based on data-oriented partitioning require one or both datasets to be indexed with a data-oriented index like the R-Tree [10].

Both Datasets Indexed

The synchronous R-Tree traversal [5] requires each dataset A and B to be indexed with an R-Tree [10] R_A and R_B. The approach starts at the root nodes of both R-Trees and synchronously traverses them to the bottom. If two nodes $n_A \in R_A$ and $n_B \in R_B$ on the same level intersect, then their children will be tested pairwise. On the leaf level, the actual elements will be tested for intersection.

By using the R-tree as a basis, the synchronous R-Tree traversal also inherits its problems. Inner node overlap and dead space in the R-Tree lead to more comparisons than necessary and therefore slows down the join. Approaches like the R*-Tree [4] or the R+-Tree [24] have been developed to reduce overlap through an improved node split algorithm or duplication. The duplication of elements, however, leads to more comparisons and to more disk accesses to retrieve the duplicates. Duplication also leads to duplicate results that have to be filtered.

If all data is known beforehand, bulkloading leads to efficient R-Trees. Several bulkloading approaches like STR [16], Hilbert [13], TGS [9] and the PR-Tree [2] have been developed, all with better performance than the R+-Tree or the R*-Tree. While TGS and PR-Tree are efficient on data sets with extreme skew and aspect ratio, Hilbert and STR perform similarly, outperforming the others on real-world data.

Synchronous index traversals can also be done when both datasets are indexed with Quadtrees [1] (or Octrees in 3D). Like in the case of the R+-Tree, however, elements are duplicated and results thus need to be deduplicated.

One Dataset Indexed

In case an index I_A exists for dataset A, the indexed nested loop join [7] loops over dataset B and for each element $b \in B$ it queries I_A to find intersecting elements. Executing a query for each $b \in B$, however, can be a considerable overhead, particularly if $B >> A$.

The seeded tree approach [18] also requires one dataset, A, to be indexed with an R-Tree I_A. The existing R-Tree I_A is used to build an R-Tree I_B on dataset B and the two R-Trees are then joined with a synchronous R-Tree traversal [5]. By building I_B based on I_A, the approach can align the bounding boxes and the synchronous join therefore has to compare fewer bounding boxes. Improvements to the basic approach use sampling to speed up building the R-Trees [17] or avoid memory thrashing [19] but still suffer from R-Tree related problems like overlap.

2.2 Space-oriented Partitioning

Space-oriented partitioning approaches assign all spatial elements to partitions. To deal with the ambiguity of the assignment, i.e., one spatial element may have to be assigned to several partitions, two classes of approaches, *multiple assignment* and *multiple matching*, have been developed.

Multiple Assignment

Multiple assignment replicates and copies each spatial element (or a reference) to all partitions it overlaps with. Replicating elements has the advantage that the spatial join only needs to compare elements in each partition with each other and not elements of different partitions. Replication, however, also has disadvantages: 1) more comparisons need to be performed and 2) result pairs may be detected twice and need to be deduplicated (during the join [6] or in the end).

The Partition based Spatial Merge join (PBSM [23]) uses a uniform grid to partition the entire space of both datasets into cells. Each element of dataset A is assigned to all cells c_A it overlaps with and all $b \in B$ are assigned to cells c_B respectively. In the next phase PBSM iterates over all pairs of cells c_A and c_B which have the same position and joins them, i.e., the elements in c_A are compared with the elements c_B to find intersections.

Multiple Matching

Multiple matching avoids replication and assigns each spatial element only to one of the partitions it overlaps with. Because an element could be assigned to one of several different partitions, all the partitions that share a border potentially must be compared with each other.

The Scalable Sweeping-Based Spatial Join [3], for example, partitions space into n equi-width strips in one dimension and assigns each $a \in A$ that entirely fits into strip n to a set LA_n (and $b \in B$ to LB_n respectively). For each n it uses a plane-sweep approach to find all intersecting pairs from LA_n and LB_n. Elements overlapping several strips, i.e., from strip j to strip k, are assigned to sets LA_{jk} and LB_{jk}. When using the plane-sweep LA_n and LB_n, all sets LA_{jk} and LB_{jk} with $j < n < k$ are also taken into account.

To use space-partitioning while at the same time avoiding replication, the size separation spatial join (S3 [14]) exploits a hierarchy of L equi-width grids of increasing granularity (in D dimensions the grid on level l has $(2^l)^D$ cells). Each element is assigned to the lowest level where it only overlaps with one cell. S3 maintains two hierarchies, H_A for A and H_B for B. To perform the join the algorithm iterates over each cell c_A of H_A and joins it with each cell of H_B that intersects with c_A on a higher level. Joining means that elements on the highest level will be compared to all other elements. The fewer elements are assigned to the highest level, the fewer comparisons will be needed.

3. MOTIVATION

GIPSY is motivated by the data management challenges the neuroscientists we collaborate with in the context of the Blue Brain Project (BBP [20]) face. We first describe the BBP, the spatial join challenges faced in it and then motivate the need for GIPSY with an experimental analysis.

3.1 Blue Brain Project

In order to simulate and understand the brain, the neuroscientists in the BBP build the most detailed and biorealistic models with data acquired in anatomical research on the cortex of the rat brain. They have started to build small models of the elementary building block of the rat neocortex, a neocortical column of about 10,000 neurons. The structurally accurate microcircuits (or models) are built on massively parallel systems (currently the BlueGene/P with 16K cores). A visualization of a small microcircuit of a few thousand neurons is shown in Figure 1 (right).

The process of building the models starts with analyzing the neurons in the real rat brain tissue in the wet lab, measuring their electrophysiological properties as well as their morphology, i.e., their shape. As Figure 1 (left) shows, the morphology of a neuron is approximated with cylinders modelling the dendrite and axon branches in three dimensions.

To build a small scale model, several hundred or thousand neuron morphologies are put together in a spatial model. Before the model can be simulated, synapses (the places where electric impulses can leap over between different neurons) need to be placed. Prior research [15] has shown that an accurate model can be built by placing the synapses where the branches (or the cylinders representing them) of different neurons intersect. More precisely, synapses are placed where a cylinder representing an axon branch and a cylinder representing a dendrite branch intersect. The process of placing synapses thus equals to a spatial join of the axon and dendrite cylinders of the neurons.

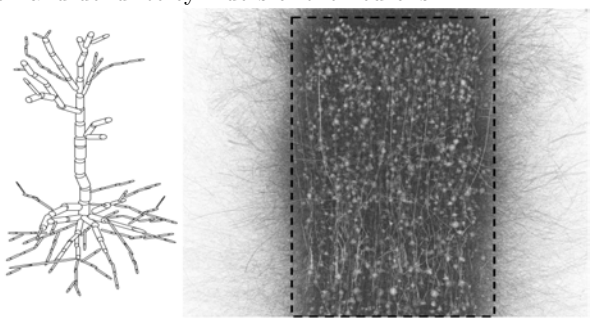


Figure 1: Schema of a neuron's morphology modelled with cylinders (left) and a visualization of a model microcircuit comprised of thousands of neurons (right).

The models currently built and simulated in the BBP contain up to 500,000 neurons with the goal to increase the size of the models many times to first simulate the brain of the rat and ultimately the human brain with $\sim 10^{11}$ neurons. More importantly, the circuits will become more detailed by modelling neurons (e.g., synapses and neurotransmitter) at the subcellular level and therefore packing orders of magnitude more spatial elements in the same space. The spatial join at the core of the model building will become more selective and its efficiency is thus pivotal.

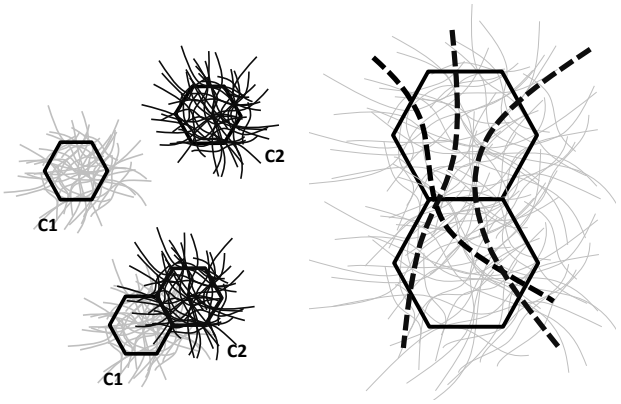
3.2 Use Cases

Currently the BlueGene/P is used to perform the spatial join on a model. The model is partitioned and loaded into the 16K cores of it and each core will perform the spatial join and then report the result. Because the memory is limited to 1GB per core, the biggest model that can be built is a column, the smallest building block of the brain featuring about 10 million neurons.

To attain the ultimate goal of simulating the entire brain, bigger models need to be built. The only way of doing so is to combine, on the disk of a single machine (or in a cluster), several columns into one big model, either by (1) combining several columns into one model or (2) connecting two columns with long ranging branches.

3.2.1 Combining Columns

Using the BlueGene/P to build the models limits the size of the biggest model to the size of the supercomputer's main memory, i.e., 10 million neurons or a column. To build bigger models, several columns need to be combined and hence the columns need to be spatially joined with each other. To speed up the join, only the branches (cylinders) penetrating the neighboring column are used for the join. Figure 2 (left) illustrates how columns C1 and C2 (view from top) are combined: only the few neuron branches (black lines) from C2 penetrating the neighboring column C1 (black lines inside C1) need to be joined with the neurons in C2 (gray lines in C2). All neurons and branches are modelled with thousands cylinders each and, in this use case, the sparse dataset (the cylinders making up the black lines from C2 inside C1) containing several hundred thousand cylinders is joined with the dense dataset (the cylinders representing the gray line of C1) containing several hundred million cylinders.



(a) Combining Columns (b) Mesocircuits

Figure 2: Illustration of the use cases.

3.2.2 Building Mesocircuits

In this use case one or few neuron branches are added to one or several columns. The added branches model the growth of mid-range fibers, i.e., model how branches penetrate a column. Also, in this case, the added branches interact with the neurons in the column, making the detection of touches between incoming branches and the rest of the circuit via a spatial join necessary. The number of cylinders added to the column in this case is typically several thousand. The sparse dataset is thus much smaller than in the previous use case, while the dense dataset contains a similar number of cylinders. Figure 2 (right) shows how the branches (dashed lines) are added to the neurons of the two columns (hexagons) to connect them.

3.3 Motivating Experiments

What is common among the aforementioned use cases is that two datasets of entirely different density, i.e., number of elements in the same space, are spatially joined. Several approaches for the spatial join of datasets have been developed, none, however, can efficiently join datasets of contrasting density. We illustrate the problem with experiments where we join a sparse dataset containing 800'000 elements with increasingly dense datasets containing between 50 and 450 million elements (20GB on disk for 450 million elements). We increase the density of the dense dataset to emulate increasingly detailed models (more elements in the same space), leading to growing overlap in indexes based on data-oriented partitioning. Both datasets contain boxes

with length 1 in each dimension placed with a uniform random distribution in a space of 1000 space units in each dimension. The description of the datasets and the setup used for these experiments is in Section 5.1.

We compare broadly used approaches like the Partition Based Spatial Merge Join (PBSM [23]), the synchronized R-Tree (R-Tree [5]) and the indexed nested loop (INL [7]). The results of the join are shown in Figure 3.3.

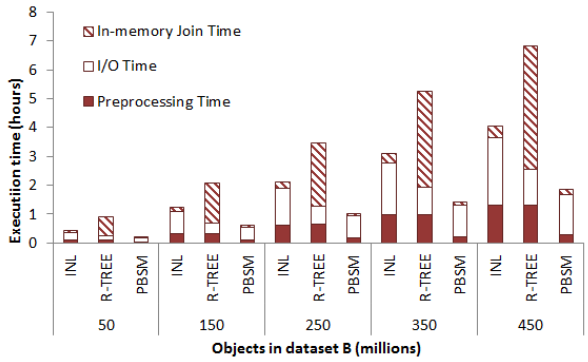


Figure 3: Total execution time as a result of joining uniform datasets of different densities.

We distinguish between three phases of each join: (1) *Preprocessing*, the time to index (or partition the data in case of PBSM), (2) *I/O Time*, the time to read partitions or disk pages into memory, and (3) the time for joining the partitions (or disk pages in case of the R-Tree) in memory, the *In-memory Join Time*.

The major problem for the data-oriented approaches, INL and R-Tree, is overlap [10]. Overlap leads to too many pages retrieved from disk and to too many (repeated and thus unnecessary) comparisons. Excessive disk reads due to overlap, however, do not result in longer execution time because many disk pages can directly be retrieved from the OS cache. Repeatedly read pages on the other hand will have to be compared multiple times, resulting in considerably longer In-memory Join Time. The R-Tree suffers more from overlap than INL because it compares all children of intersecting nodes of both R-Trees pairwise, resulting in a substantially higher In-memory Join Time.

Space-oriented approaches like PBSM, on the other hand, suffer from the coarse-grained partitioning. Configuring PBSM is difficult: choosing the grid too coarse-grained leads to big partitions and consequently to too many elements compared pairwise when joining partitions. Setting it too fine-grained, i.e., small partitions leads to excessive replication, resulting in more comparisons and consequently a slower join. The configuration used (25^3 partitions) for the experiment is the best we were able to identify with a parameter sweep. Still, because the sparse dataset typically has at least one element in each partition, the entire dataset has to be retrieved for the join, resulting in excessive I/O.

4. THE GIPSY APPROACH

With GIPSY we want to overcome the problems of approaches based on data-oriented as well as space-oriented partitioning. As outlined previously, space-oriented approaches cannot partition the dense dataset fine-grained enough so that the join can only retrieve the data needed from disk. Partitioning more fine-grained results in small partitions, therefore more replication and consequently a slower join. Data-oriented approaches, on the other hand, suffer from overlap resulting in unnecessary pages retrieved from disk followed by unnecessary comparisons.

4.1 Overview

The novelty of GIPSY lies in avoiding the coarse-grained partitioning of space-oriented approaches and using the fine-grained data-oriented partitioning. At the same time it avoids the excessive disk page reads and comparisons because of overlap in the tree structure of data-oriented approaches. Instead of traversing a tree structure top down, GIPSY cleverly traverses the data itself using a crawling approach [22, 25].

More precisely, GIPSY indexes the dense dataset and takes the elements of the sparse dataset and visits them one after the other by *walking* between them using the index on the dense dataset. Once it arrives at the location of a particular element e of the sparse dataset, it uses *crawling* to detect all elements of the dense dataset that intersect with e and then walks to the next element. Using an index on the dense dataset and traversing it directed by the elements of the sparse dataset makes the join particularly efficient for the repeated join of a dense dataset with multiple sparse ones. Figure 4 illustrates how GIPSY uses the sparse dataset to direct walking in the dense dataset.

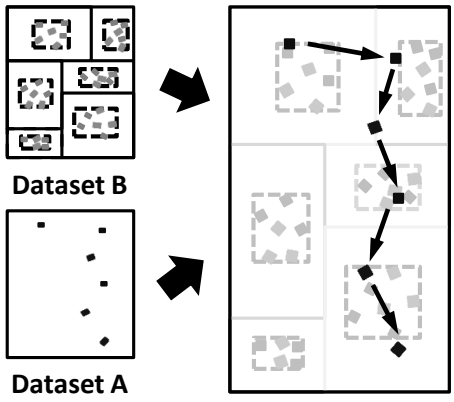


Figure 4: GIPSY uses the sparse dataset to walk/crawl through the dense dataset.

To enable GIPSY's novel approach of combining data-oriented partitioning with crawling to execute a spatial join, we need an efficient method to partition the dataset data-oriented, and to add & store the information needed for walking/crawling. Additionally, we need an effective method to find a start element for the walk as well as an order in which the elements of the sparse dataset can be visited with minimal distance between them.

In the following we discuss the methods, algorithms and data structures needed.

4.2 Indexing the Dense Dataset

Unlike mesh datasets indexed with DLS [22], the datasets we use (spatial datasets in general) do not have any inherent connectivity information like mesh edges. In GIPSY we therefore first partition the dataset and then store information needed for crawling in additional data structures, similar to other crawling approaches [25].

Algorithm 1 Partitioning Algorithm

Input: elements: all spatial elements of B
partitionsize: number of elements in each partition

Output: partitions: resulting partitions
Data: x_partitions: set of partitions
y_partitions: set of partitions

calculate number of partitions in each dimension $pn = \sqrt[3]{size(elements)/partitionsize}$

sort *elements* on x-coordinate
make *pn* partitions of consecutive x-coordinate values
insert partitions into *x_partitions*

foreach *partition* $p \in x\_partitions$ **do**
  sort p on y-coordinate
  make *pn* partitions of consecutive y-coordinate values
  insert partitions into *y_partitions*

  foreach *partition* $p \in y\_partitions$ **do**
    sort p on z-coordinate
    make *pn* partitions of consecutive z-coordinate values
    insert partitions into *partitions*

  end
end
return *partitions*

4.2.1 Partitioning the Dense Dataset

To partition the dense dataset we use an approach similar to Sort-Tile-Recursive (STR) [16], a method initially designed for bulkloading R-Trees. While we are not interested in the R-Tree it produces, its approach to data-oriented partitioning is useful so that spatially close elements can be stored on the same disk page, thereby preserving spatial locality.

Similar to STR, GIPSY first sorts the dense dataset on the x-dimension of the element center and partitions the elements along this dimension. All resulting partitions are sorted on the y-dimension and partitioned again. Finally, the resulting partitions are also sorted on the z-dimension and partitioned. The process is illustrated with pseudocode in Algorithm 1.

Figure 5 illustrates the partitioning. The solid lines represent the partition MBRs (minimum bounding rectangle), whereas the dashed lines represent the elements MBRs that wrap more tightly the actual spatial elements.

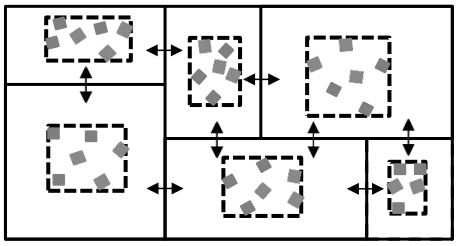


Figure 5: Partioning of the dataset with solid lines for the partitions and dashed lines for the elements MBRs.

By choosing the size of the partitions at every step of the partitioning process, we can precisely determine the size of the final partitions. This (a) ensures that a partition fits on a disk page and (b) gives us a parameter to control the granularity of the partitioning. In GIPSY, the size of the partitions is always chosen so that it fits on a disk page, i.e., of size 4K or a multiple. GIPSY stores the elements in each partition together on a disk page called elements page.

4.2.2 Crawling Information

In order for GIPSY to work, we need to determine and store the information that enables walking/crawling. The relevant information are the partitions and their neighborhood relation, i.e., what partition neighbors what other partitions.

Storing the Crawling Information

For each partition we store the minimum bounding rectangle (MBR) of the elements and the partition. The elements MBR is the minimum bounding box containing all elements on a page whereas the partition MBR is the minimum bounding rectangle of the partition. Most importantly, we also need to store the neighbors of each partition. We store all information in *summary records*: each record summarizes a partition p and stores a pointer to the elements page of p, p's partition MBR, p's elements MBR, as well as the neighbors of p (the partitions intersecting with p or touching p).

Determining the Crawling Information

The information of the partitions follows directly from the partitioning process. We determine the neighbors by performing a spatial self-join on the partition MBRs. This computes, for each partition p, what partitions n neighbor (touch or intersect) p. Any spatial join method can be used for the self join. Nevertheless, in GIPSY we use PBSM because we identified it as the quickest method to perform a one-off spatial join.

As Figure 6 shows, all summary records are stored on disk pages called summary pages. We store as many summary records on a summary page as possible. When retrieving a summary record, it is likely that neighboring ones (spatially close ones) will also be retrieved and preserving spatial locality will thus improve performance. As a consequence, we use the Hilbert space filling curve [11], calculate the Hilbert value of each summary record (of the center of its partition MBR) and store on the same summary page summary records with consecutive Hilbert values. On the summary pages we do not store the partition identifier as a neighbor but instead store the identifier of the summary page it is stored. Such an approach simplifies and speeds up the join process as no mapping (correlating the summary page identifier with the disk page identifier) needs to be queried repeatedly.

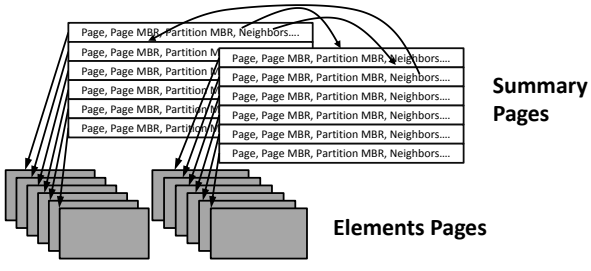


Figure 6: The data structures of GIPSY: summary pages, elements pages and pointers between them (arrows between summary records).

Algorithm 2 Directed Walk

Input: startrecord: start crawl record
a_k: spatial element of sparse dataset A_i
Output: closestrecord: closest summary record to a_i

closestrecord = startrecord;
while *(distance(closestrecord.elementsMBR, a_k) > 0 AND !chekgettingaway())* **do**
  records = read all neighbor records of *closestrecord*;
  foreach *summary record r ∈ records* **do**
    if *distance(r.elementsMBR, a_k) < distance(closestrecord.elementsMBR, a_k)* **then**
      closestrecord = r;
    end
  end
end
return *closestrecord*

4.3 Joining the Datasets

To finally join the datasets, GIPSY takes a sparse dataset A_i (without indexing it) and iterates over all its elements $a \in A_i$. It uses the start summary record at the beginning of the join and walks in the dense dataset to find the spatial location of the first element a_1 of the sparse dataset. For that matter it uses a directed walk: it recursively reads all neighboring summary records and picks the one closest to a_1 (smallest distance of the elements MBR to a_1). As Algorithm 2 illustrates with pseudocode, this process is repeated until a summary record intersecting with a_1 is found. If no neighbor record closer to a_1 can be found and the elements MBR of the closest record still does not intersect with a_1, then a_1 does not intersect with any element from B.

Once an *intersection record*, a summary record of which the elements MBR intersects with the element a_1, is found the directed walk ends and the crawl phase starts. The goal of the crawl phase is to find all elements of the dense dataset intersecting with a_1. Starting with the *intersection record*, the crawl phase, similarly to the walk phase, recursively visits all neighbors until no more element intersecting with a_1 can be found. More precisely, it starts with the *intersection record* and recursively retrieves all summary pages that contain referenced neighbor records. If the elements MBR of a summary record intersects with a_1, then the elements page is retrieved and all elements are tested for intersection. If the partition MBR of a summary record does not intersect, then the neighbors are not visited and hence the crawl phase ends when no more crawl record with a partition MBR intersecting with a_1 can be found. Algorithm 3 illustrates the crawl phase with pseudocode. If no summary record of which the elements MBR intersects with a_1 can be found, then a_1 does not intersect with any element and we walk to the next element in A.

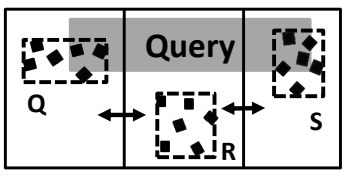


Figure 7: Starting with partition Q, GIPSY has to recursively visit all neighbors with intersecting partition MBRs.

The algorithm also illustrates why GIPSY needs to store and use both, the partition and elements MBRs. The elements MBR is needed in the join process to determine whether or not to retrieve an elements page (if the query intersects with the elements MBR). The partition MBR, on the other hand, is needed to guarantee correctness: given a partition (and its summary record) Q, even if the elements MBR of Q's neighbor R does not intersect with the query, R's neighbor S elements MBR might. Consequently GIPSY cannot stop visiting R's neighbors only because its elements MBR does not intersect with the query, but only if the partition MBR does not intersect as Figure 7 illustrates.

As the pseudocode in Algorithm 4 shows (using Algorithm 2 as directedWalk and Algorithm 3 as crawl), after finding all elements of the dense dataset that intersect with a_1, the same process, i.e., directed walk and then crawling, is repeated to find the intersections of the following elements a_k until all elements of A_i intersecting with elements of the dense dataset B are identified.

Algorithm 3 Crawl Algorithm

Input: intersectionrecord: crawl record with page MBR intersecting with $a \in A_i$;
range: MBR of spatial element $a \in A_i$
Output: result: spatial elements
Data: squeue: summary record queue
visitedqueue: already visited elements queue

enqueue *intersectionrecord* into *squeue*

while *squeue* $\neq \emptyset$ **do**
  dequeue summary record *s* from *squeue*
  if *s.elementsMBR intersects with range* **then**
    retrieve elements page *ep* referenced in *s*
    foreach *element* $\in$ *p* **do**
      if *element MBR intersects range* **then**
        put *element* into *result*
      end
    end
  end
  if *s.partitionMBR intersects with range* **then**
    foreach *neighbor in s* **do**
      if *neighbor is not in visitedqueue* **then**
        enqueue *neighbor* summary record in *squeue*
      end
    end
  end
end
return *result*

Algorithm 4 GIPSY Join

Input: elements: array of spatial elements A_i
startrecord: summary record from where to start join
Output: elements: set of elements A_i intersecting with element from B
Data: intersectionrecord: summary record holding the current intersection record
intersectingelements: elements $\in B$ intersecting with an element $in A_i$

intersectingrecord = startrecord;
foreach *element a ∈ elements* **do**
  intersectionrecord = directedWalk(*a*, *intersectingrecord*);
  intersectingelements = crawl(*intersectionrecord*);
  add all *intersectingelements* to *elements*;
end
return *elements*

4.4 Visiting Order

The order in which the elements of the sparse dataset are visited has an impact on the distance walked and consequently also on the execution time of the join. While walking a longer distance and retrieving more summary pages does not automatically need to translate into more time needed to access the disk (due to caching of the OS), walking longer will, however, mean more time is spent on comparing summary records to elements of the sparse dataset.

An ideal visiting order minimizes the overall distance walked, similar to the travelling salesman problem (TSP). Unfortunately, the TSP is NP-hard and we have to resort to heuristics to find an order that approximates the optimal order in reasonable time. We have implemented several strategies to sort the sparse dataset and evaluate them in the experimental section in Section 5.

4.5 Start Point

To visit the elements of the sparse dataset, GIPSY needs to start at a particular summary record of the dense dataset and walk through it. GIPSY could start with a random (or chosen by some heuristic) summary record, use a directed walk to the closest summary record of a_1 (the first element of the sparse dataset) and then start the join process. This method, however, depends on the randomly chosen summary record as well as the sparse dataset and may thus involve an infeasibly long walk.

To reduce the distance between the start point and the first element of the sparse dataset, we index all summary records of the dense dataset. Any spatial index could be used to index the dense dataset so that a first summary record close to an element of the sparse dataset can be retrieved. To avoid the issue of overlap and also to speed up the process of building the index we refrain from using an R-Tree or related spatial indexes. Instead we calculate the Hilbert value of each summary record (the Hilbert value of the center of the elements MBR) and index them with a B+-Tree.

To find the summary record to start from, we execute the range queries (the Hilbert values of the sparse dataset elements) on the B+-Tree in order to find the first intersection, i.e., the summary record with the closest Hilbert value to one of the elements of the sparse dataset. This summary record does not necessarily contain the first element of the sparse dataset but will be spatially close to it and GIPSY will walk to it and start the traversal there.

The B+-Tree can also be reused in case of an extremely sparse dataset: instead of an infeasibly long walk between two elements a_i and a_{i+1} of the sparse dataset, it may be more efficient to use the B+-Tree instead to find a summary record close to a_{i+1}. In our experiments, however, we have not encountered a dataset where using the B+-Tree repeatedly improves performance.

5. EXPERIMENTAL EVALUATION

In this section we describe the experimental setup & methodology, compare GIPSY against state-of-the-art spatial join approaches and analyze its performance. We use real neuroscience datasets and, to study the impact of dataset characteristics on the performance and to make the experiments reproducible, synthetic datasets where we control number, size and distribution of the elements.

5.1 Setup

Hardware: The experiments are run on Red Hat 6.3 machines equipped with 2 quad CPUs AMD Opteron, 64-bit @ 2700 MHz, 4 GB RAM and 4 SAS disks of 300GB (10000 RPM) capacity as storage. We only use one of the disks for the experiments, i.e., no RAID configuration is used.
Software: All algorithms are implemented single-threaded in C++ for a fair comparison.
Setting: We experimentally compare the Indexed Nested Loop Join (INL), Synchronized R-Tree Traversal (R-TREE), Partition Based Spatial Merged Join (PBSM) and our approach - GIPSY. R-TREE and PBSM use the plane sweep algorithm as the in-memory join.

Due to the absence of appropriate heuristics, we set the parameters of related approaches optimally after a parameter sweep. In case of PBSM we found the configuration with 25^3 partitions to be the most efficient. This configuration provides the best trade-off between the number of elements needed to be compared by the plane sweep algorithm and the number of elements replicated, deduplicated, additionally written/read to/from disk. INL and R-TREE have shown the best performance with a fanout of 135.

The disk page size in all experiments is 8 KB. Experimental conditions assume a cold file system cache, i.e., after the preprocessing/indexing step (PBSM: partitions creating, assigning elements; R-Tree based approaches: index building; GIPSY: partitioning the space, introducing neighborhood information, B+-Tree building) OS caches and disk buffers are all cleared.

5.2 Experimental Methodology

We use two different types of datasets in the experiments: (1) to control the dataset characteristics (number, size and distribution of elements) and demonstrate general applicability we use synthetic datasets and (2) to demonstrate the impact on our use cases we also use neuroscience datasets.

Synthetic Datasets: We create synthetic datasets by distributing spatial boxes in a space of 1000 space units in each dimension of the three-dimensional space. The length of each side of each box is determined with uniform random distribution between 0 and 1. Spatial elements are distributed in space depending on the data distribution.

We use three different data distributions - uniform, normal ($\mu = 0$, $\sigma = 220$) and clustered and always join datasets of the same distribution. For the clustered dataset we choose uniformly randomly centers of the clusters in the three-dimensional space and place between 500 to 1000 spatial elements around the cluster center using a normal distribution ($\mu = 0$, $\sigma = 220$). The number of spatial elements in the datasets is between 10K and 450M. The corresponding size on the disk is between 468KB and 20GB.

Neuroscience Datasets: To evaluate GIPSY on real data we use a small part of the rat brain model represented with 450 million cylinders as elements. We take from this model a contiguous subset with a volume of 285 μm^3 and approximate the cylinders with minimum bounding boxes. In the spatial join process axons are represented by one dataset, dendrites by the other and the detected intersections represent synapses. The number of spatial elements in the datasets is between 10K to 250M. The corresponding size on the disk is between 468KB and 11GB.

Approach: Like most spatial join methods we focus on

the filtering step [12], i.e., finding pairs of spatial elements whose approximations (MBRs) intersect. The refinement step of the join, detecting the intersection between the actual shape of the elements, is a computationally hard problem with little room for improvement. We do not consider the refinement step in the experimental evaluation.

In all experiments we fix the size of dataset A (sparse dataset) and gradually increase the size of dataset B (dense dataset). We increase the density of the dense dataset to emulate the increasingly detailed models the neuroscientists build. To build more biorealistic models, they increase the number of elements in the same space, i.e., they increase the density, leading to growing overlap in indexes based on data-oriented partitioning.

Inspired by the use cases from Section 3.2 we evaluate GIPSY with two different types of experiments where the density of the sparse dataset differs. Based on the use case in Section 3.2.2 we use sparse datasets that contain only a few thousand spatial elements for a first set of experiments. The second set of experiments is inspired by Section 3.2.1 and uses sparse datasets with 800× more spatial elements (several hundred thousand elements).

In all experiments we measure the total execution time and the number of I/O operations during the spatial join process. The total execution time is measured for two different scenarios, for a one-time operation (joining dataset A with B) or for a repeated spatial join operation (joining several datasets A_i with one B).

We break the total execution time in preprocessing time, I/O time and in-memory join time. The preprocessing time is the time necessary to build the initial data structures (PBSM: partitions creating, assigning elements; R-Tree based approaches: index building; GIPSY: partitioning the space, introducing neighborhood information, building the B+-Tree). The I/O time is time spent on data writing/loading during the join process and the in-memory join time is the time needed to join data in memory, i.e., comparing the spatial elements and related operations.

In case of the repeated spatial join we reuse the index (partitions for PBSM, tree for R-TREE and INL) which is created during the first spatial join process and reused thereafter. The total execution time for these experiments thus contains the preprocessing time only once.

5.3 Synthetic Datasets

We evaluate GIPSY on synthetic data with two sets of experiments both inspired by the two neuroscience use cases described in Section 3.2.

5.3.1 Combining Columns

In the following set of experiments we fix the size of dataset A to 800K and join it once with datasets B of increasing size from 50M to 450M, in steps of 100M. All datasets have a uniform distribution. Figure 8 shows the total execution time broken down into preprocessing time, I/O time and in-memory join time. In this experiment GIPSY outperforms all other algorithms and its improvement over PBSM, the fastest state-of-the-art approach, is between 16% - 25%.

The performance of state-of-the-art data-oriented approaches, i.e., R-tree based approaches, is degraded due to overlap. In case of R-TREE the overlap problem is more evident and affects primarily the in-memory join (it also affects I/O but this is not obvious because the OS caches disk pages). The R-TREE approach is affected by overlap in two R-Trees.

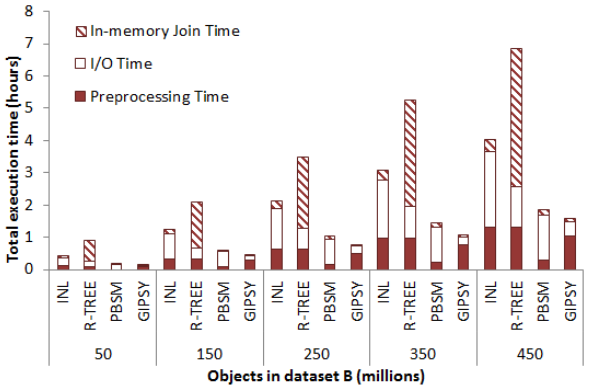


Figure 8: Total execution time as a result of one spatial join, combining columns.

Overlap at higher levels in each tree means that more R-Tree nodes overlap between the trees and consequently their children are compared pairwise, leading to a considerably bigger number of comparisons which ultimately results in an increase of the in-memory join time.

INL essentially repeatedly executes a small range query on the R-Tree for every element in A. The in-memory join time hence does not grow considerably with increasing overlap because every inner node of the R-Tree retrieved during query execution only has to be compared against the range query (unlike R-TREE where due to overlap all children of overlapping nodes need to be compared with each other). Overlap in INL, however, means that more nodes and thus disk pages need to be read (in a tree without overlap, a range query can be executed by accessing as many nodes as the tree is high). The I/O time, however, does not grow considerably because the OS caches disk pages.

PBSM is the fastest state-of-the-art approach, but as a space-oriented partitioning it has to retrieve all data. The I/O time of PBSM consequently makes up most of the execution time. Most of the preprocessing phase is spent assigning elements to the partitions and because these writes can be done sequentially, the preprocessing phase does not take significant time. Disk accesses to read the partitions, on the other hand, are mostly slow random reads, resulting in substantial I/O time.

In comparison to other algorithms GIPSY spends significantly less time on comparisons and I/O operations (I/O and in-memory join time). On average 60% of the total execution time is spent on the preprocessing step.

In the remaining experiments we primarily compare GIPSY with PBSM, the fastest state-of-the-art approach.

In the next experiment we join 10 different sparse datasets of 800K elements with the same dense dataset containing 450M spatial elements. The total execution time is the sum of all spatial joins and includes the time for building the index on the dense dataset B just once (created during the first spatial join and reused thereafter). As Figure 9 shows for repeated joins, after 10 spatial joins, GIPSY already outperforms PBSM by a factor of 2.62.

As the previous experiments showed, the main problem of PBSM is unnecessary data retrieval when joining datasets of different densities. In the next experiment we measure the unique disk pages read during the spatial join when we increase the density of the dense dataset B. As Figure 10

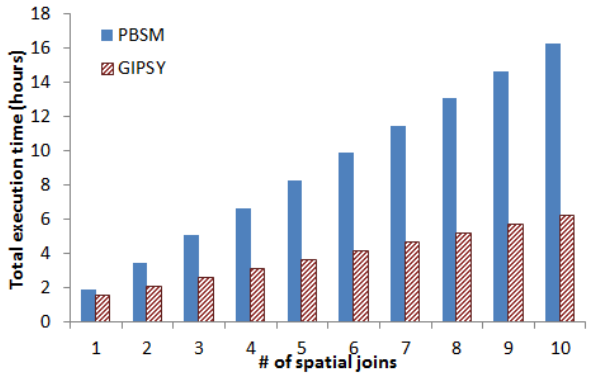


Figure 9: Total execution time as a result of repeated join, combining columns.

shows, with increasing density of B also the ratio between PBSM and GIPSY I/O increases: for example, in the case of joining uniform datasets of 800K with 450M elements, PBSM needs to read 2.71 times more unique disk pages than GIPSY. GIPSY reduces the data read from disk by using fine-grained data-oriented partitioning.

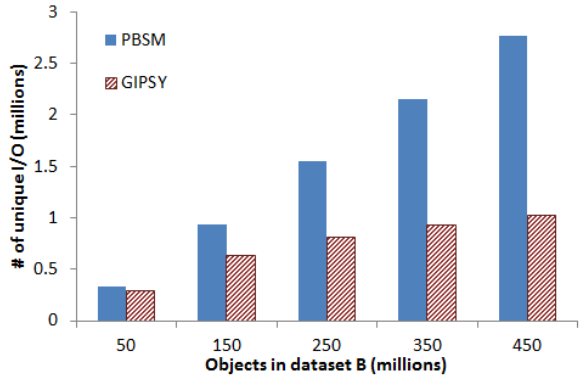


Figure 10: Number of I/Os as a result of one spatial join, combining columns.

5.3.2 *Building Mesocircuits*

Inspired by the second use case described in Section 3.2.2, in this set of experiments we join one (or several) very sparse datasets with a dense dataset. We decrease the size of the sparse dataset by a factor of 800 so it contains 10K spatial elements but use the same experimental methodology as before (increasing the dense dataset B from 50M to 450M).

The results of these experiments are shown in Figure 11. Compared to the previous experiments only INL differs in relative performance as its execution is slower compared to the R-TREE. The relative performance of the R-Tree-based approaches, however, has improved and if we take into account index reuse, i.e., we do not consider the preprocessing time, PBSM is now slower than all other algorithms.

R-Tree-based approaches perform better than PBSM (in a case of the index reuse) because they essentially are a compromise between (data-oriented) fine-grained partitioning and the overlap problem. The overlap in the index built on the dense dataset remains the same but because of the fine-grained partitioning, only the few disk pages containing

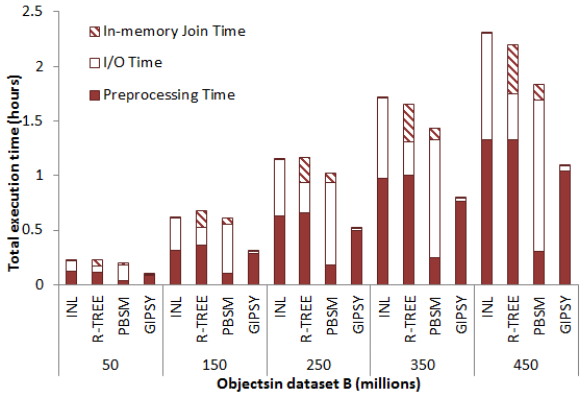


Figure 11: Total execution time as a result of one spatial join, building mesocircuits.

the data needed can be retrieved, resulting in much less accesses to the disk. The in-memory join phase (time spent on comparing node, elements etc.) and the overall time needed to traverse the whole R-Tree is significantly reduced for R-TREE and INL as well.

PBSM, on the other hand, still needs to retrieve all of the dense dataset. Figure 12 illustrates the total number of I/O operations (logscale) executed during the join process of PBSM and GIPSY (without the preprocessing phase). In case of joining two datasets with 10K and 450M elements PBSM reads 54.48× more unique pages from disk.

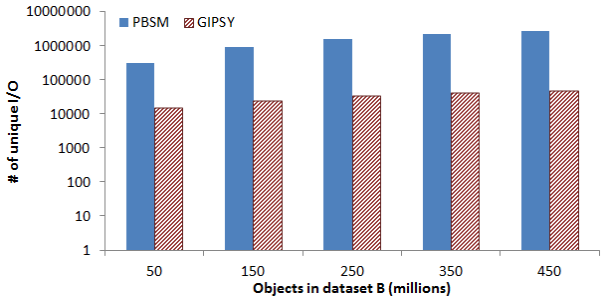


Figure 12: Number of I/Os (logscale) as a result of one spatial join, building mesocircuits.

Based on the total execution time GIPSY achieves the best results with an overall improvement of 2.35 compared to PBSM. During a single spatial join GIPSY, however, spends on average 90% of total execution time in the preprocessing phase building the index. Not considering the preproceessing phase, i.e., if the index is reused for repeated joins, GIPSY achieves total speedup up to 17.95 compared to the R-TREE, the best known approach when joining repeatedly.

The experimental results comparing GIPSY with R-TREE when repeatedly joining datasets are shown in Figure 13. We join 10 different sparse datasets each containing 10K spatial elements with one dense dataset of 450M elements. The total execution time is the sum of all previous spatial joins and includes the time necessary for the preprocessing step only once. When joining all 10 sparse datasets with the dense dataset, GIPSY attains a speedup compared to the synchronized R-Tree traversal of 6.78. The total execution time, in the case of GIPSY, appears to be a flat line because it increases minimally with each spatial join.

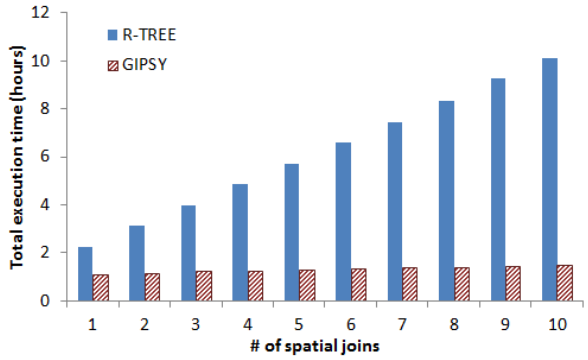


Figure 13: Total execution time as a result of repeated join, building mesocircuits.

5.4 Neuroscience Datasets

As a litmus test and to demonstrate the usefulness of GIPSY for the neuroscientists, we also test its performance on neuroscience datasets. We use a similar methodology as before and set the size of sparse dataset A to 450K spatial elements (dendrites) for the use case described in Section 3.2.1 and to 10K for the use case described in Section 3.2.2. The dense dataset B is increased from 50M to 250M (axons).

Figure 14 shows the results of one time spatial join for both use cases. The R-TREE approach is significantly slower than in previous experiments because the data distribution leads to more overlap in both R-Trees. Our experiments on neuroscience data show that more inner nodes need to be retrieved for both R-Tree-based approaches as compared to the spatial join executed on the uniform datasets (increase in the inner node reads from 2.34 for 50M to 3.27 for 250M) which is a good indication for increased overlap.

In the case of the repeated spatial join we compare GISPY to the second best approaches, i.e., PBSM (for the combining columns use case) and to INL (for the mesocircuit use case). The results in Figure 15 show a speedup of 3.5 (compared to PBSM) for the repeated join of the combining columns use case and of 2 (compared to INL) for the repeated join of the mesocircuit use case.

5.5 GIPSY Sensitivity Analysis

In the following we analyze the impact of sort strategy, page size and data distribution on the performance of GIPSY.

5.5.1 *Impact of Visiting Order*

In the walking phase GIPSY walks through the dense dataset directed by the elements of the sparse dataset. The order in which the elements of the sparse dataset are visited should ensure that the overall distance is as small as possible, as walking between the elements results in disk accesses (to read crawl pages) and in comparisons.

In the following, we evaluate the impact of different visiting strategies on the performance of GIPSY. We execute a spatial join between a sparse uniform dataset with 800K spatial elements and a dense uniform dataset with 100M elements. We compare four different sort strategies on the sparse dataset: none (use dataset as it is), a nested loop sort, X-axis sort and Hilbert sort. The X-axis sort sorts the elements based on their x-coordinate while the Hilbert sort sorts based on the Hilbert value [11] of the center of the element. The nested loop sort compares all elements pairwise and visits them in the order of minimal pairwise distance. In each step we exclude the elements for which we have already found the minimum distance from further consideration.

The results of this experiment are shown in Figure 16 where we divide the total execution time into sort time and join time. Due to the long sort execution time we exclude the nested loop sort from the experiment. The time necessary to sort the sparse dataset, in case of X-axis and Hilbert sort, is insignificant. GIPSY's performance is degraded by a factor of 4.8 when not using any sort strategy and 2.4 times when sorting on the x-dimension only. Hilbert sort requires, like the other approaches, almost no time to sort and ultimately performs the join the fastest.

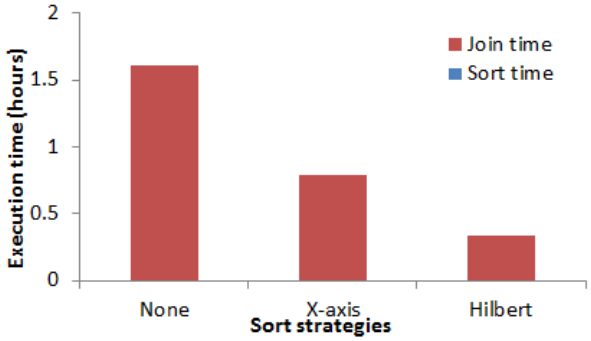


Figure 16: Impact of different sort strategies on GIPSY performance.

5.5.2 *Impact of the Data Distribution*

In the following experiments we measure the impact of the data distribution on GIPSY by running the experiments from Section 5.3.1 on datasets with uniform, clustered and Gaussian distribution. Figure 17 and Figure 18 show the experimental results, i.e., the spatial join time and number of detected intersections. GIPSY's join is divided into: Seeding - time necessary to obtain start point, Walking - walk time and Crawling - the crawling phase.

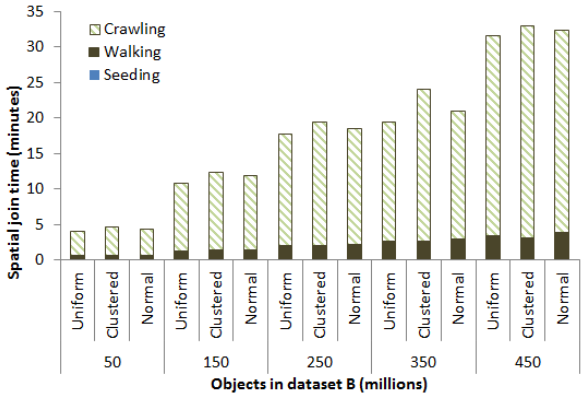


Figure 17: Spatial join time for uniform, clustered and Gaussian data distribution.

The overall spatial join time does not vary significantly for the three different distributions. GIPSY takes slightly more time for joining clustered data, followed by Gaussian and uniform data. This difference in performance can be explained by the selectivity (Figure 18).

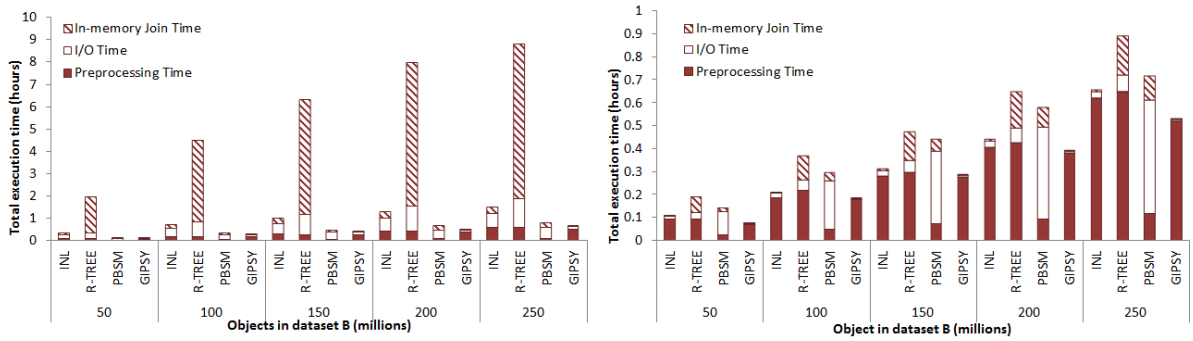


Figure 14: Total execution time as a result of one spatial join for neuroscience datasets, combining columns (left) and building mesocircuits (right).

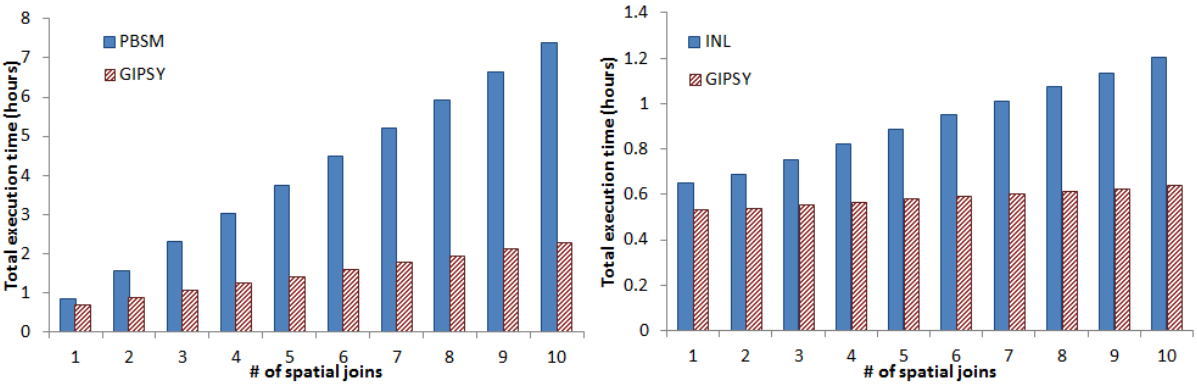


Figure 15: Total execution time as a result of repeated join for neuroscience datasets, combining columns (left) and building mesocircuits (right).

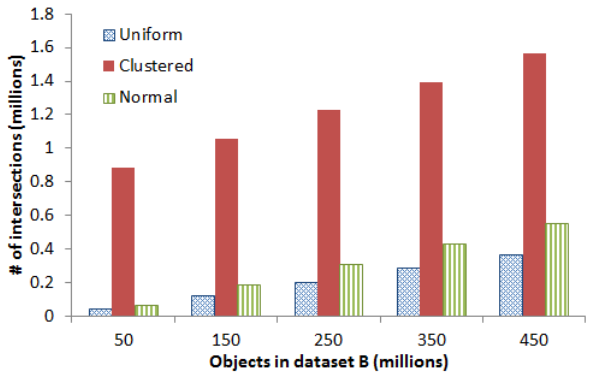


Figure 18: Number of intersections, uniform, clustered and Gaussian data distribution.

As Figure 17 shows, GIPSY in general spends significantly less time on the directed walk compared to the crawling phase. This was the initial assumption for developing GIPSY - we rely on spatial element proximity and ensure to walk as little as possible by following a particular visiting order. The crawling phase, on the other hand, depends heavily on the average number of neighbors (the denser the dataset is, the more neighbors we have to examine). The time needed to find a start point is insignificant in all cases.

5.5.3 *Impact of Page Size*

As discussed, GISPY's performance depends on the number of neighbors the summary records stored on the summary pages have. If the summary pages are bigger (stored on bigger disk pages) they contain more records to examine. In order to measure its impact on GIPSY we execute the experiments from Section 5.3.1 (combining columns use case), varying the size of page from 4KB to 64KB.

Figure 19 shows the result of the experiment. GIPSY's join time is divided into: walking I/O - time spent on retrieving neighborhood information in the walking phase, walking - walking related operations (e.g., distance calculations), crawling I/O - time spent on I/O operations executed during crawling phase, crawling - crawling related operations (e.g., overlap detection).

A change in the page size is a trade-off. Increasing the page size, on the one hand, leads to fewer neighbors per summary record and fewer random reads. At the same time one node contains more data, i.e., more summary records that need to be examined. Because the element pages contain more elements, their page/partitions MBRs increase, and consequently we have more unnecessary data-retrieval/ comparisons. The results of the join between 800K and 450M for page sizes of 4KB and 64KB confirm our expectation: the time spent on I/O operations decreases while the time spent on crawling and walking related operations increases. In our experiments, the best performance for dense datasets

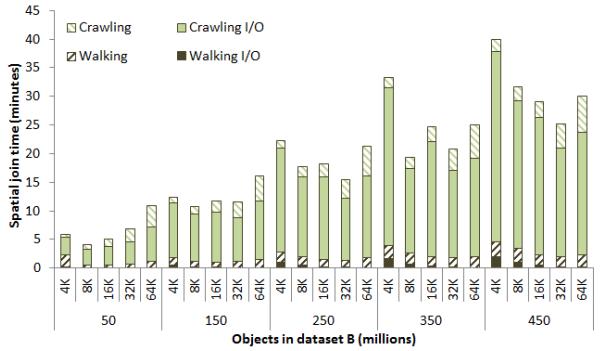


Figure 19: Spatial join time, varying the page size from 4KB to 64 KB.

is obtained for a page size of 32KB.

6. CONCLUSIONS

In this paper we identify the problem of joining datasets of contrasting density, i.e., joining several sparse datasets with a dense dataset. State-of-the-art approaches do not join these datasets efficiently. Data-oriented approaches suffer from overlap resulting in excessive reads from disk and in unnecessary comparisons. Space-oriented approaches cannot partition the datasets fine-grained enough and typically the entire dataset has to be read from disk although only a small part is needed.

The novelty of GIPSY, the approach we develop to tackle the challenge, lies in the efficient combination of crawling with data-oriented partitioning to join spatial datasets. GIPSY indexes the dense dataset with a data-oriented approach and avoids overlap through crawling: the sparse datasets are used to crawl through the index of the dense dataset. Only the small parts of the dense dataset needed for the join are retrieved.

In our experiments we show the effectiveness of GIPSY as it outperforms state-of-the-art disk-based spatial join algorithms between a factor of 2 & 18 and is particularly efficient when joining a dense dataset with several sparse datasets. We have tested GIPSY on neuroscience but also on synthetic datasets, demonstrating that it can be efficiently used on spatial datasets from other domains/applications as well.

7. REFERENCES

[1] W. G. Aref and H. Samet. Cascaded Spatial Join Algorithms with Spatially Sorted Output. In *International Workshop on Advances in Geographic Information Systems (AGIS '96)*.

[2] L. Arge, M. D. Berg, H. Haverkort, and K. Yi. The Priority R-tree: A Practically Efficient and Worst-case Optimal R-tree. *ACM Transactions on Algorithms*, 4(1):1–30, 2008.

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

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD '90*.

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

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

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

[8] 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.

[9] Y. J. Garcia, M. A. Lopez, and S. T. Leutenegger. A Greedy Algorithm for Bulk Loading R-trees. In *International Workshop on Advances in Geographic Information Systems (AGIS '98)*.

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

[11] D. Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38:459–460, 1891.

[12] E. H. Jacox and H. Samet. Spatial Join Techniques. *ACM Transactions on Database Systems*, 32(1):7, 2007.

[13] I. Kamel and C. Faloutsos. On Packing R-trees. In *CIKM '93*.

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

[15] J. Kozloski, K. Sfyrakis, S. Hill, F. Schurmann, 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.

[16] S. T. Leutenegger, M. A. Lopez, and J. Edgington. STR: a Simple and Efficient Algorithm for R-tree Packing. In *ICDE '97*.

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

[18] M.-L. Lo and C. V. Ravishankar. Spatial Joins Using Seeded Trees. In *International Conference on Management of Data (SIGMOD '94)*.

[19] N. Mamoulis and D. Papadias. Slot Index Spatial Join. *IEEE Transactions on Knowledge and Data Engineering*, 15(1), 2003.

[20] H. Markram. The Blue Brain Project. *Nature Reviews Neuroscience*, 7(2):153–160, 2006.

[21] S. Nobari, F. Tauheed, T. Heinis, P. Karras, S. Bressan, and A. Ailamaki. TOUCH: In-Memory Spatial Join by Hierarchical Data-Oriented Partitioning. In *SIGMOD '13*.

[22] S. Papadomanolakis, A. Ailamaki, J. C. Lopez, T. Tu, D. R. O'Hallaron, and G. Heber. Efficient Query Processing on Unstructured Tetrahedral Meshes. In *SIGMOD '06*.

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

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

[25] F. Tauheed, L. Biveinis, T. Heinis, F. Schürmann, H. Markram, and A. Ailamaki. Accelerating range queries for brain simulations. In *ICDE '12*.

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