Speeding Up Range Queries For Brain Simulations

Farhan Tauheed[†‡], Laurynas Biveinis[¶], Thomas Heinis[†], Felix Schürmann[‡], Henry Markram[‡], Anastasia Ailamaki[†]

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

[¶]*Department of Computer Science, Aalborg University, Denmark*

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

***Abstract*— Neuroscientists increasingly use computational tools to build and simulate models of the brain. The amounts of data involved in these simulations are immense and efficiently managing this data is key.**

One particular problem in analyzing this data is the scalable execution of range queries on spatial models of the brain. Known indexing approaches do not perform well, even on today's small models containing only few million densely packed spatial elements. The problem of current approaches is that with the increasing level of detail in the models, the overlap in the tree structure also increases, ultimately slowing down query execution. The neuroscientists' need to work with bigger and more importantly, with increasingly detailed (denser) models, motivates us to develop a new indexing approach.

To this end we developed FLAT, a scalable indexing approach for dense data sets. We based the development of FLAT on the key observation that current approaches suffer from overlap in case of dense data sets. We hence designed FLAT as an approach with two phases, each independent of density.

Our experimental results confirm that FLAT achieves independence from data set size as well as density and also outperforms R-Tree variants in terms of I/O overhead from a factor of two up to eight.

I. Introduction

Scientists in various disciplines increasingly use computational tools to simulate, process and analyze experimental data. Computational tools make it substantially simpler for them to conduct scientific tasks. At the same time however, scientists are also increasingly buried in the data deluge produced by their tools. Being able to access the relevant parts of their data, i.e., their spatial models, quickly in order to analyze, understand, and prepare new experiments is pivotal for them.

In this paper we thus develop a new index that efficiently supports scientists in executing range queries on dense data sets stemming from increasingly detailed spatial models.

The work presented in this paper is motivated by our collaboration with the Blue Brain Project (BBP [17]). With data acquired in anatomical research on the cortex of the rat brain the neuroscientists in the BBP build biophysically realistic models, the most detailed computer models of the brain to date, for simulation based research in neuroscience. The project began by focusing on the elementary building block of the neocortex, a neocortical column of about 10,000 neurons. Morphologically speaking, each of these neurons has branches extending into large parts of the tissue in order to receive and send out information to other neurons. Figure 1 (left) shows a cell morphology, with cylinders modeling the branching of the dendrite and axon in three dimensions.

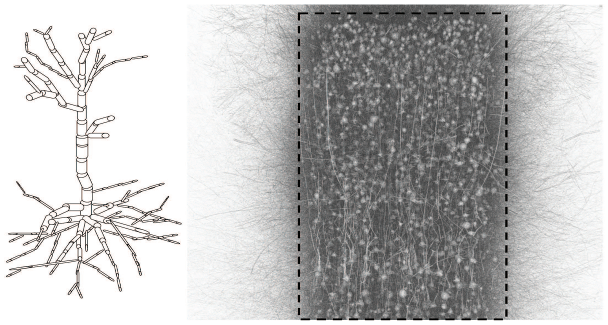


Fig. 1: Schema of a neuron's morphology modeled with cylinders (left) and a visualization of a model microcircuit comprised of thousands of neurons (right).

The BBP targets both building as well as simulating brain models. The use of massively-parallel computing in the BBP (BlueGene/P with 16k cores) nowadays allows the placement of several thousands of neurons in three dimensions to recreate and simulate structurally accurate microcircuits [14]. An example microcircuit of a few thousand neurons is shown in Figure 1 (right).

The microcircuits currently investigated in the Blue Brain Project contain up to 500,000 neurons, but it is foreseen to ultimately extend this by orders of magnitude to microcircuits of the size of the human brain ($\sim 10^{11}$ neurons). More importantly, the circuits will at the same time become much more fine grained, modeling the neurons at a subcellular level with hundreds of thousands of structural elements per neuron.

To support neuroscientists in analyzing spatial brain models we want to develop an indexing approach that executes range queries independent of the size and density of the brain model. Achieving scalability for range queries is not just a problem in neuroscience, but also in many other scientific disciplines [8], [13] which use dense spatial data sets. During the development of FLAT we make very few assumptions about the data sets so that it can also be used for data sets with similar characteristics. However, because the models in our use cases change only slowly, if at all, we focus on developing a bulkloading approach and do not consider updates.

The remainder of the paper is structured as follows. We discuss related work and its shortcomings in Section II. In Section III we analyze in more detail two use cases and show how state-of-the-art approaches do not scale for our data sets.

In Section IV we give a high level overview of FLAT. Section V first discusses how we index the brain data and in Section VI we explain how to evaluate a range query on the data. We compare the approach to related approaches and benchmark it in Section VII. Finally, we use FLAT to index other data sets, report on the results in Section VIII and conclude in Section IX.

II. Related Work

Database research has produced many approaches to spatial indexing [6] in recent years.

A first class of spatial indexes are point access methods such as the KD-Tree [4] and the Quadtree [21], along with its variant for 3D space, the Octree [11]. While these approaches are mainly used in memory, two extensions of the KD-Tree also work on disk, the KDB-Tree [5] and the Bkd-Tree [20]. The latter claims a better performance in the case of updates. These indexing techniques can, however, only be used to index points and not spatial objects. Because our data set contains volumetric elements, we would need to duplicate elements which occupy several partitions on the leaf level, thereby increasing the index size several times.

To index connected tetrahedral meshes, crawling approaches like DLS [19] use approximate search algorithms coupled with walking [19] but require the data set to be convex and to contain connectivity information. While our data sets contain connected objects (neuron branches), they also contain concave regions, i.e., ’holes’. Concave regions can split the connected data set inside a range query into two parts, preventing the algorithm from crawling from one part to the other. Known crawling approaches hence do not work for our data.

Arguably the seminal data structure is the R-Tree [9]. The R-Tree is a disk-based, multi-dimensional generalization of the B-Tree [2], which recursively encloses objects in minimum bounding rectangles (MBRs). Several extensions of the basic approach have been proposed. They can generally be divided into two classes: bulkloaded R-Trees, where the data set is known a priori, and R-Trees where elements are inserted consecutively.

The R*-Tree [3] falls in the latter category and reduces overlap through an improved version of the node split algorithm along with the removal and reinsertion of spatial elements (once a node overflows). The R+-Tree [22] similarly tries to avoid overlap through the duplication of the MBRs. Doing so however leads to a much bigger index.

Several packing methods for the R-Tree have been proposed to bulkload data sets which are known a priori. With these, spatially close elements can be packed on the same disk page in order to improve locality and to reduce overlap between nodes. The Hilbert R-Tree [12] uses the Hilbert space filling curve to order the objects. Consecutive elements in this order are spatially close and are hence packed on the same page. Sort-Tile-Recursive (STR) [16] recursively tiles the space, sorts the elements in a tile along each dimension and thereby also guarantees spatial proximity as well as small MBRs. With this, STR outperforms the Hilbert R-Tree [16]. As opposed to STR, the Top down Greedy Split (TGS) [7] works top down: it splits the data set into partitions so that on each level the area of each partition is minimized. This process continues recursively until each partition fits on a disk page. While bulkloading with TGS takes much longer than with other approaches, the resulting R-Tree outperforms the Hilbert R-Tree and STR on extreme data sets (extreme with respect to skew, aspect ratio).

To improve handling of extreme data the Priority R-Tree (PR-R-Tree) [1] has been proposed. It groups all elements with extreme coordinates in the same dimension in the same node, thereby reducing the area and overlap of the remaining nodes. This improves its performance on synthetic and extreme data sets, making the PR-RTree outperform TGS.

Despite the numerous improvements and approaches to alleviate the problem of overlap in the R-Tree, it still introduces considerable I/O. The denser a data set is, i.e., the more spatial elements are in the same unit of space, the more overlap the bounding boxes of the R-Tree have, and hence the bigger the I/O overhead of query execution becomes.

III. Motivation & Use Cases

Because the spatial data sets modeling the brain are both very dense and concave, readily available indexing approaches cannot be used. Using bounding box based indexing techniques such as R-Tree based approaches [9] introduces substantial I/O overhead.

We illustrate this by indexing brain data sets of increasing density with different bulkloading variants of the R-Tree. To increase density we keep the volume the same but gradually add elements to the model. We measure the overlap in the first (Hilbert [12]), the most commonly used (STR [16]), and the most recent (PR-Tree [1]) bulkloaded R-Tree by executing point queries at random locations. The point query is an excellent indication of overlap in an R-Tree: the number of disk pages read to execute this query in an R-Tree without overlap is equal to the height of the tree. The more disk pages need to be read, the more overlap the R-Tree has.

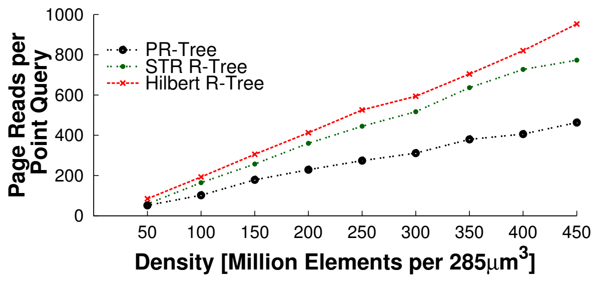


Fig. 2: Point query performance on R-Tree variants.

The results in Figure 2 clearly show that with increasing density, the overlap also increases: the height of the tree is 5, yet the number of pages retrieved for a single point query grows to more than 450 for the densest data set indexed with the PR-Tree. The overlap problem in R-Trees has a substantial impact on the range query performance in the following two neuroscience use cases.

A. *Structural Neighborhood*

The structural connectivity in the BBP microcircuits, i.e. the places where two neuron branches touch (and where electrical impulses may hence leap over) is currently precomputed [14]. While this allows to build the circuits fast, it is not well suited to detect proximity incrementally, i.e., detect the proximity of fibers from other brain regions.

Data Set Density in Million Elements per 285μm³	50	100	150	200	250	300	350	400	450
Number of Page Reads Per Result Element	1.73	1.85	1.94	1.87	2.1	2.13	2.24	2.28	2.33

Fig. 3: Number of page reads per result for structural neighborhood range queries on a Priority R-Tree.

To compute proximity incrementally, numerous requests for the immediate neighborhood, e.g. all elements within a distance of 5μm ($5 \times 10^{-7}\%$ of the space), along a neuron fiber are executed in sequence.

Because R-Trees built on such dense data sets have considerable overlap, executing these queries has a substantial I/O overhead. To show this we index a brain model with a bulkloaded Priority R-Tree and measure the number of page reads per result element for very small random range queries (each covering $5 \times 10^{-7}\%$ of the space). While keeping the volume constant, we increase the number of elements in the model and therefore the density.

The results in Figure 3 show that with a data set of 450 million elements the average number of page reads for each element in the result set (of 56'000 elements) is 2.33. This is already a substantial overhead and the trend indicates that this approach will not scale with increasing density of the data set. Clearly we have to find a more scalable approach to support this use case better.

B. *Large Spatial Subvolumes*

For the purpose of visualization and analyses, e.g., tissue density, specific subvolumes need to be retrieved with range queries. The requested subvolume sizes are typically big and can go beyond $5 \times 10^{-4}\%$ of the data set space.

In the case of queries with increasing query volumes, the number of overlapping minimum bounding rectangles (MBRs) enclosing a query decreases, and hence fewer ambiguous paths need to be followed in the R-Tree. The I/O due to overlap thus decreases and overlap no longer dominates the overhead. At the same time however, in order to retrieve a large result set, the number of non-leaf pages which need to be read grows substantially. The non-leaf pages in the R-Tree are the pages which store hierarchy information as opposed to the leaf pages which store the spatial elements.

We measure this overhead with an experiment showing the total number of Gigabytes retrieved when querying the R-Tree variants with 200 random queries (each of a size of $5 \times 10^{-4}\%$ of the space) and compare it with the size of the actual result set in Gigabytes. As Figure 4 shows, the difference is significant, but more importantly, the ratio between the size of the result and the size of the data the PR-Tree (the best

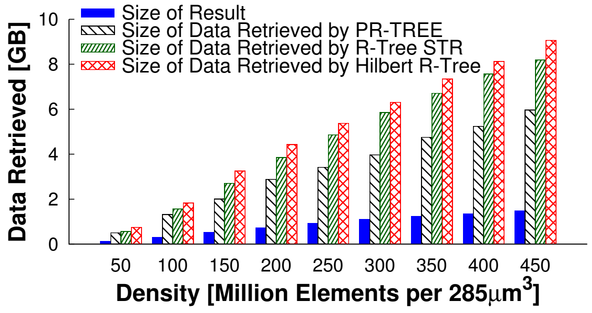


Fig. 4: Total bytes read and result set size for large spatial subvolume queries on R-Tree variants.

R-Tree in this experiment) retrieves grows from 3 to 4 with the density of the data set. This overhead is mainly due to the non-leaf pages which have to be read, i.e., organizing the data hierarchically. Clearly we have to find an approach that scales better.

C. *Contributions*

The contribution of this paper is the development of FLAT, an indexing approach that scales for increasingly dense data sets of the brain.

We base the development of FLAT on the key observation that, given any start element inside a query range, recursively visiting its neighbors inside the range only depends on the number of elements in the result (and does not suffer from overlap). However, to be sure that all elements inside the query range are visited, the data set needs to have information regarding what elements neighbor each other. This leaves us with two key challenges that we address in the development of FLAT.

First, we develop an efficient mechanism to find a random start element inside the query range. The second, more difficult challenge is to compute what elements neighbor each other in the data set, i.e., neighborhood information. We thus develop an efficient method to compute neighborhood information along with data structures from where to retrieve it efficiently. With this, FLAT scales for data sets of increasing density and tackles the problems of the two use cases, i.e., it avoids overlap and retrieval of an excessive number of non-leaf nodes.

Albeit the ideas FLAT is based on are simple, they prove to be very effective. We improve the query execution time for the previously described two use cases, between a factor of 2 and 4 for the first and between a factor of 2 and 6 for the latter, compared to the fastest R-Tree variant. In further experiments we also show performance improvements on other scientific data sets. The trends of our results strongly indicate that FLAT will scale substantially better for more detailed brain data sets used by the neuroscientists in the near future.

IV. The FLAT Approach

FLAT indexes spatial elements of arbitrary shape and uses these elements as primary keys to retrieve further information (such as electrical properties etc.) about them. Similarly to R-Tree variants, FLAT wraps elements of arbitrary shape in axis aligned minimum bounding rectangles (MBR) and stores both, the actual element and its MBR. To evaluate a query FLAT tests, like the R-Tree, whether the MBR of an element intersects with or is contained in a range query.

On a high level, FLAT uses the following data structures and two phased query execution approach. Elements and their MBR are stored as follows: spatially close elements are stored on the same page on disk. For each page of elements neighborhood information, i.e., pointers to other spatially close pages are also stored. In particular, if disk page A contains an element which is close to another element on page B, then we store a pointer from A to B. Additionally, for each page the minimum bounding rectangle (the page MBR) enclosing all elements on it is stored together with a reference to the page. We use a traditional spatial index to index the page MBRs and refer to it as the seed index. This concept, with rectangles representing groups/pages of spatial elements and arrows expressing neighborhood, is illustrated in Figure 5.

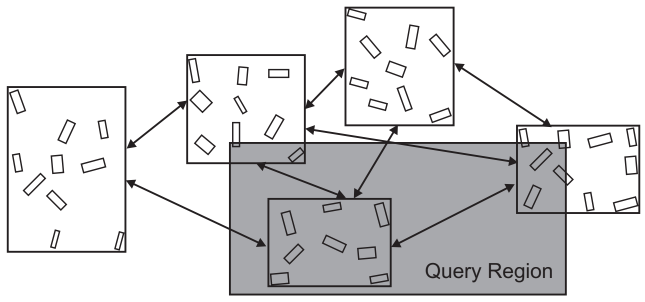


Fig. 5: FLAT: Spatially close elements are packed on the same disk page (rectangle) and pointers (arrows) are added between neighboring pages.

Given the seed index and the neighborhood information, the result of range queries is computed in two phases:

- **Seed Phase:** Given a query range, an arbitrary page MBR intersecting this range is retrieved from the seed index. The key insight we use for this phase is that retrieving an arbitrary element in a given range is a cheap operation, even with an R-Tree. It does not suffer from overlap: instead of having to follow all paths, only one single path has to be followed from the root of the tree to one of the leafs. The complexity of this operation is typically in the order of the height of the R-Tree.
- **Crawl Phase:** With the page MBR retrieved in the seed phase, a reference to the corresponding disk page is also obtained. This page is retrieved and all spatial elements on it are tested if they are in the query region. Following this, its neighbor pages are retrieved recursively until no more elements in the query range are found.

This approach ensures that evaluating a range query no longer depends on the density of the data. The complexity of the seed phase is in the order of the height of the tree and the crawl phase depends on the size of the result set. At the same time, the approach does not need to retrieve hierarchically stored information.

Because the data sets in our use cases, as is true for many scientific data sets, change infrequently and updates happen in batches, reindexing is more efficient, and we hence do not consider updates. Instead, and also because our data sets are known a priori, we develop FLAT as a bulkloading approach.

V. FLAT Indexing

At the core of FLAT indexing is the algorithm to compute the neighborhood information that ultimately allows it to crawl through (possibly unconnected) data sets. In the following section we first discuss the algorithm to compute the neighborhood information, and then the data structures needed to store this information.

A. Computing Neighborhood Information

To compute the neighborhood information we segment the entire space into partitions, with each partition containing one page. For a page P and the partition A it is contained in, we consider a page as a neighbor of P if its partition is adjacent to or overlaps with A. For FLAT to work, we compute all neighbors of a page P, P's partition MBR (the MBR enclosing the entire partition A) and P's page MBR (the MBR enclosing all the elements stored on P). Figure 6 shows an example partitioning with the partition MBRs (dashed lines) and the page MBRs (solid lines) enclosing the elements stored on a page. The neighbor relationships between pages are illustrated with arrows.

The exact procedure used to partition the space does not matter in order for FLAT's query evaluation to work as long as the resulting set of partitions satisfies two properties. First, the partition procedure must not leave any empty space, i.e., the union of all partitions must cover the entire space. Second, each partition MBR must enclose the MBR of the corresponding page.

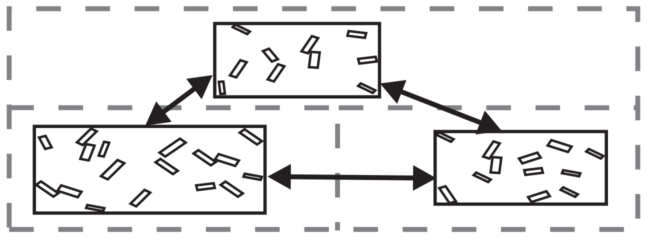


Fig. 6: Partitioning procedure used to determine page MBRs (solid lines) as well as partitions (dashed lines).

We use an efficient algorithm based on STR [16] (see pseudocode in Algorithm 1) to first sort the spatial elements on the x-dimension and then to partition them along this dimension. Each resulting partition is again sorted and partitioned based on the centers of the y-dimension. The same procedure is repeated for the third dimension. The partition sizes in each of the partitioning steps are chosen so that the final partitions contain at most as many spatial elements as can be stored on one disk page. Once all partitions are calculated, the partition MBRs and page MBRs are calculated. To make sure the partition properties discussed before are satisfied,

Algorithm 1 FLAT Indexing Algorithm

```
Input: elements: array of all spatial elements
       size: number of spatial elements in data set
       pagesize: number of elements on each disk page
       rtree: R-Tree seed index
Output: partitions: array of all resulting partitions
Data: xpart: set of partitions
      ypart: set of partitions
      zpart: set of partitions
      partitions: set of partitions

calculate number of partitions in each dimension $pn = \sqrt[3]{size/pagesize}$

sort $elements$ on x-coordinate of element center
make $pn$ partitions of consecutive x-coordinate values
insert partitions into $xpart$

foreach partition $p \in xpart$ do
    sort elements $\in p$ on y-coordinate of element center
    make $pn$ partitions of consecutive y-coordinate values
    insert partitions into $ypart$

    foreach partition $p \in ypart$ do
        sort elements $\in p$ on z-coordinate of element center
        make $pn$ partitions of cons. z-coordinate values
        insert partitions into $zpart$

        foreach partition $p \in zpart$ do
            calculate page MBR $pageMBR$ of $p$
            calculate partition MBR $partitionMBR$ of $p$
            stretch $partitionMBR$ to contain $pageMBR$
            store $page$ and $partition$ MBR in $rtree$
            insert $p$ into $partitions$
        end
    end
end
foreach partition $p \in partitions$ do
    retrieve $neighbors$ partitions in $rtree$ intersecting with $partitionMBR$ of $p$
    store $p$'s $neighbors$, $pageMBR$ and $partitionMBR$ together with $p$ in $partitions$
end
return partitions
```

each partition is stretched so that it encloses the MBR of the corresponding page. Following this, all partition MBRs are inserted into a temporary R-Tree, used solely to compute the neighborhood information. Finally, for each partition, a range query with the partition MBR is executed, and all intersecting partitions, the neighbors, are retrieved.

B. Data Structures

The indexing algorithm returns a set of partitions, and for each partition the spatial elements in it, the page MBR, the partition MBR, and the neighbors. In the following section we discuss how this information is stored in order to support efficient query execution with FLAT. We particularly discuss FLAT's core data structures, the *seed index* used to retrieve a start element, *metadata* used to retrieve neighborhood information, and finally *object pages* used to store the actual data, the spatial elements.

1) Seed Index: To start the search, the *seed index* must return an arbitrary element inside the query range. Finding an arbitrary spatial element inside a range is in many spatial indexes independent of the density. In the R-Tree for example, despite the overlap due to the data set density, only one of several ambiguous paths through the tree needs to be followed. The number of nodes visited/disk pages read is thus in the order of the height of the R-Tree.

In FLAT we use an R-Tree to find a starting point for the crawl phase. In this *seed index* we index each page MBR (the MBR of an object page), along with a pointer to the disk page (object page). We modify the implementation of the R-Tree to retrieve intersecting page MBRs along with its object page until it finds an object page on which one element is in the query range. Once such a page is found, querying the seed index is stopped and the page is used as a starting point. In the rare case of nearly or completely empty queries, several leaf nodes may need to be visited until a page MBR is found. If no object page can be found, then the query has no result.

2) Metadata & Neighborhood Information: Storing the neighborhood information on the same page P as the elements is difficult, because depending on what elements are on P, a different number of pointers needs to be stored on it. The number of pointers that need to be stored cannot be known a priori, and resorting to ad-hoc mechanisms, e.g., to reserve a certain amount of space, leads to underfilled pages, defeating the goal of storing as many elements as possible on one page. We therefore store the neighborhood information and the spatial elements separately.

The neighborhood information is stored in an additional data structure referred to as metadata. FLAT stores per object page one metadata record summarizing it, i.e. the record contains a pointer to the object page, the page MBR, the partition MBR, as well as pointers to the neighbors (to their metadata record) of the object page.

When retrieving one metadata record, it is likely that its spatially close neighbors are retrieved as well. To improve performance we need to preserve the spatial locality of the metadata records and therefore store them in the leafs of the seed tree, i.e., we index each record R with R's page MBR as key and R as value. The neighborhood information stored in the records consequently contains pointers to the other records inside the R-Tree leafs. Storing the records in the leafs of the seed tree (an R-Tree) ensures that spatially close records are stored on the same leaf page. At the same time, we fit as many records as possible on each leaf to make good use of disk space.

All data structures and their relations are illustrated in Figure 7: several metadata records are stored on each of the leafs in the seed index. Neighborhood information is stored as pointers from one metadata record to another, possibly between different leaf nodes. Each metadata record also contains a pointer to an object page.

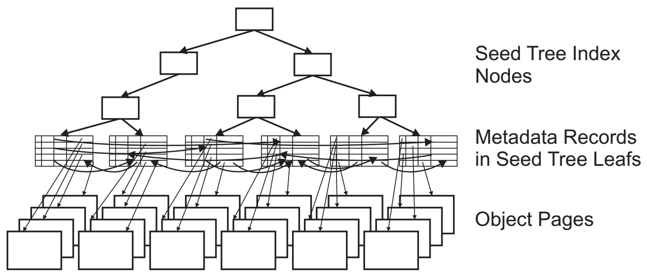


Fig. 7: The three data structures and their interaction: seed tree, the metadata records in its leafs point to each other and to the object pages.

3) Object Pages: To optimize the fill factor of the object pages we pack the maximum number of elements (the exact number depends on the size of the structural elements, e.g., for a mesh triangle 9 floats/doubles suffice, for an MBR/axis aligned box it is 6 floats/doubles) on each disk page.

Also on the object pages we want to preserve spatial locality, i.e., store spatially close objects on the same page. The partitions STR produces preserve spatial locality better [16] than Z-order [18] or Hilbert-packing [12]. Because Algorithm 1 is based on STR, we can directly use its output (the partitions) of Algorithm 1 to determine what elements to store together on a disk page.

VI. FLAT QUERYING

FLAT uses a breadth first search to crawl the connected neighborhood of a start page. More precisely, FLAT finds a metadata record in the seed index that points to a page containing a spatial element which intersects with the query range. It follows the neighborhood pointers to other records stored in other leafs of the R-Tree. The neighbor pointers stored in a metadata record M are only followed if M's partition MBR intersects with the query. Similarly, the object page is only read from disk if M's page MBR intersects with the query. The final output of the breadth first search is the set of spatial elements which intersect with the query region. The algorithm is more formally described in Algorithm 2.

Algorithm 2 also shows that the choice of the start page during the seed phase affects neither the accuracy nor efficiency of the search. Choosing a different starting element only changes the order in which the pages are visited. Because the algorithm keeps track of what pages have been visited, each page is visited at most once.

Intuitively, evaluating a query can be stopped if no more neighbors with a page MBR intersecting the query can be found. If however this condition is used, the query in a case such as the one depicted in Figure 8 cannot be properly evaluated. This example illustrates why FLAT needs to store and use the partition MBR. Using the shaded page on the left of Figure 8 as the start page, query evaluation would already stop after looking at the first two neighbors: none of them has a page MBR intersecting with the query region. FLAT

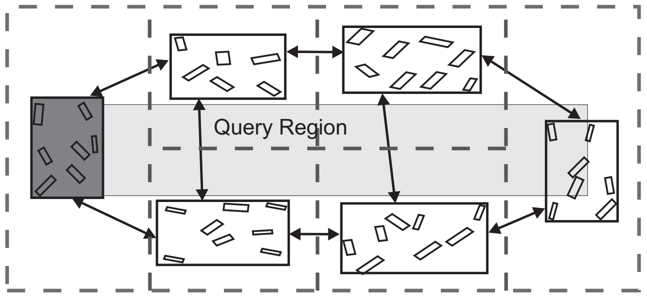


Fig. 8: Partitions intersecting the query range must be considered during the breadth first search.

Algorithm 2 Breadth-First Search Algorithm

```
Input: mr: pointer to metadata record from seed index;
       rq: range query
Output: result: set of spatial elements
Data: queue: metadata record queue;
      visited: set of visited object pages
put $mr$ into $queue$
while $queue \neq \emptyset$ do
    dequeue metadata record $m$ from $queue$
    if object page $p$ referenced by $m \notin visited$ then
        if pageMBR of $m$ intersects with $rq$ then
            retrieve object page $p$ referenced in $m$
            insert $p$ into $visited$
            foreach $element \in p$ do
                if $element$ MBR intersects $qr$ then
                    put $element$ into $result$
                end
            end
        end
        if partitionMBR of $m$ intersects with $rq$ then
            foreach $neighbor$ in $m$ do
                enqueue $neighbor$ metadata record in $queue$
            end
        end
    end
end
return result
```

therefore uses the partition MBR of a page P and checks if it intersects with the query region to decide if the neighbors of P are visited. If it does intersect, then its neighbors are tested.

The situation illustrated in Figure 8 also gives an intuition as to why a partitioning needs to have the two properties discussed in Section V-B.2 in order to work with the breadth first search:

- **Empty Space:** if we allowed empty space between partitions, no neighborhood pointers would be inferred and hence the breadth first search would not continue across such empty spaces.
- **Partition MBR encloses Page MBR:** if the partition does not enclose the page MBR, then situations may

occur where the page MBR intersects with the query range, but the partition MBR does not. Other partition MBRs then cover the page MBR but effectively prevent the page from ever being read. Figure 9 illustrates this situation where partition MBR A (dashed line) does not entirely cover the page MBR A (solid line). If the breadth first search starts at page and partition C, it will continue to partition B but go no further because partition MBR B does not intersect with the query range. A will never be read.

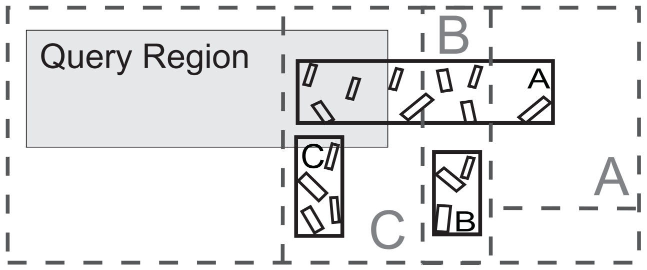


Fig. 9: A scenario where the query is not evaluated properly because the partition does not cover the page MBR.

VII. Experimental Evaluation

In the following section we describe the experimental setup & methodology, measure the performance of FLAT and study the impact of data set characteristics on FLAT. We compare FLAT against several bulkloaded R-Tree variants because bulkloaded trees outperform other R-Tree variants such as the R*-Tree [3], primarily due to better page utilization. Because the models of the brain change only slowly and the changes occur as batches, reindexing with a bulkloading approach is simpler and more time efficient than updating.

A. Setup

Experimental Setup
The experiments are run on a Linux Ubuntu 2.6 machine equipped with 2 quad CPUs AMD Opteron, 64-bit @ 2700MHz and 4GB RAM. The storage consists of 4 SAS (10’000 RPM) disks of 300GB capacity each, striped to a total of 1TB.

We use a readily available implementation of the STR R-Tree [10] but adapt so it stores 85 elements on the leaf level. Additionally, we use our own implementations of the Hilbert R-Tree [12] as well as Priority R-Tree (PR-Tree) [1] to compare against FLAT. All approaches store data on the disk in 4K pages. The seed index also uses 4K nodes. The fill factor for the R-Tree variants and the seed tree is set to 100%. All implementations store 85 spatial elements on a 4K page.

For all the experiments the OS can use the remaining memory to buffer disk pages. For a fair comparison the implementations of all approaches are single threaded.

Experimental Methodology
For the measurements we use data sets that model a small part of the brain with cylinders as spatial elements. The model contains 100’000 neurons in a volume of $285 \mu m^3$. For both use cases presented in Section III, the neurons are modeled with 450 million cylinders (an example neuron modeled with cylinders is shown in Figure 1 left). Each cylinder is described by two end points and a radius for each endpoint. For a fair comparison, we only store the MBRs of the cylinders on R-Tree leaf pages and on the FLAT object pages. All approaches therefore test if the range query intersects with the MBRs stored (axis aligned minimum bounding rectangles). We use double precision floating point numbers to represent the coordinates of the MBRs.

Through experimentation and modeling by the neuroscientists, the brain model perpetually grows and at the same time becomes more fine grained. To determine the trend of FLAT and the R-Trees, we progressively increase the density of the data set in each experiment by adding more neurons to the same volume, i.e., 50 million more cylinders in every step. With this we can extrapolate how FLAT performs for more dense data sets of the brain model in the future.

Inspired by the two use cases described in Section III we define two micro-benchmarks. The SN benchmark is derived from the structural neighborhood use case and consecutively executes 200 spatial range queries each with a fixed volume of $5 \times 10^{-7}\%$ of the entire data set volume. The LSS benchmark is derived from the large spatial subvolume use case and also consecutively executes 200 spatial range queries, but each with a fixed volume of $5 \times 10^{-4}\%$ of the entire data set. The location and aspect ratio of all queries is chosen at random. Before each query is executed, the OS caches and disk buffers are cleared (overwritten with an empty file).

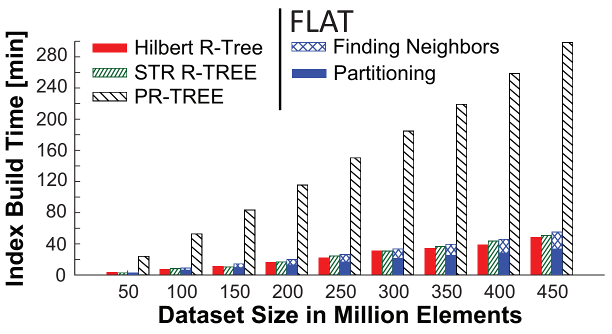


Fig. 10: Overall time to index for data sets of increasing density.

B. Time to Index

Indexing with the Hilbert R-Tree is simple: each element needs to be assigned a Hilbert value, the entire data set is sorted once on this value and the tree is built recursively. Building the Hilbert R-Tree should therefore be very fast.

Both the STR-bulkloaded R-Tree and FLAT use STR to pack the elements on pages. In doing so they sort the entire data set in three steps of recursion in each dimension. Packing the elements on disk therefore takes the same time for both approaches. Because FLAT also needs to find the neighbors of each partition we expect that will take longer to index.

The Priority R-Tree builds the tree by sorting the data set in one dimension (on either the low or high coordinate in this dimension), identifying the extreme elements (as many elements as fit on one page) in each dimension, splitting the remaining elements into two subsets and recursively applying the same procedure on the two subsets in the next dimension. The extreme elements identified in every step in each dimesion are stored together in the same leaf node of the tree. Once all leaf nodes are written, the same procedure is recursively applied to the next higher level of the tree. This entire procedure sorts the data several times recursively in each dimension and thus takes quite a long time.

The results of the experiment shown in Figure 10, where we index data sets of increasing size, confirm our expectations: to index the same data set FLAT takes longer than Hilbert, slightly longer than STR but is substantially faster than the PR-Tree. Optimizations for bulkloading the PR-Tree speed up indexing considerably [1]. But even with these optimizations, the PR-Tree remains slower than STR & FLAT because all data needs to be sorted at least six times, twice in each dimension (once on the minimum and once on the maximum coordinate).

Indexing with FLAT is rather fast, and because indexing only needs to be done very infrequently, once the model of the brain changes, the cost of it is acceptable. More importantly however, because the trend of FLAT is linear, we can expect it to scale to denser data sets.

C. *Index Size*

Both the R-Tree variants and FLAT store 85 spatial elements on a disk page. The total size of the leaf pages of the R-Trees is thus the same as the size of FLAT's object pages. Also the non-leaf pages of the R-Trees and the seed tree pages of FLAT need the same space to be stored. The only difference is that FLAT additionally stores the metadata in the seed tree. The FLAT index is hence bigger than the R-Trees as the experiment in Figure 11 confirms.

In this experiment we have compared the size of the index for data sets of increasing density for FLAT and the PR-Tree. We only compare against the PR-Tree because the different bulkloading strategies only pack the elements differently in quality but not in quantity, i.e., they pack different elements on the same page but always the same number. The space required, as well as the ratio between non-leaf and leaf pages, is exactly the same for all variants.

The most important point this experiment demonstrates however is that the size of the total index predominantly depends on the number of elements, and that we can hence expect that it grows linear with increasing density of the data set.

D. *Use Case Benchmarking*

In the following experiments we compare the execution of the SN and LSS benchmarks with FLAT and the R-Tree variants regarding the number of disk pages read, as well as the time taken to execute the queries.

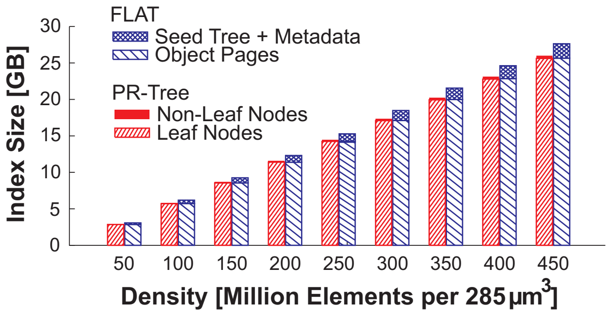


Fig. 11: Index size for data sets of increasing density.

SN Benchmark Results

Time & Page Read Comparison

FLAT does not suffer from problems related to density (like overlap) and hence scales better for dense data sets. As can be seen in Figure 12, which shows the number of page reads required to execute the benchmark, the amount of data read to compute a query with FLAT is substantially smaller than with any of the R-Tree variants. The best R-Tree, the PR-Tree, retrieves 8 times more data from disk than FLAT for the densest data set (450 million elements in $285\mu m^3$).

This experiment also confirms previous results [16] showing that STR bulkloading outperforms Hilbert bulkloading. It also corroborates results [1] indicating that the PR-Tree requires less page reads than other bulkloaded R-Trees on extreme data.

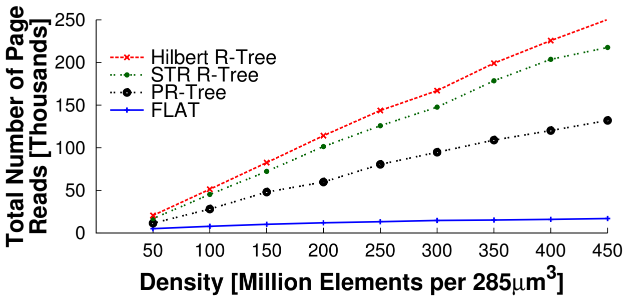


Fig. 12: Total number of page reads when executing the SN benchmark.

Comparing the shape of the curves in this experiment with the next measurement in Figure 13, where we measure the execution time of queries, it becomes obvious that the execution time for queries with FLAT and the R-Tree variants is I/O bound. More importantly however, the trends for execution time and page reads indicate that FLAT scales linear for data sets of increasing density.

Overhead Analysis

By measuring the information read from leaf and from non-leaf pages for data sets of increasing density in Figure 14 (right) we can corroborate the assumption that the R-Tree (we use the best R-Tree, the PR-Tree, as an example in this experiment) suffers from overlap. In this experiment the number of both types of pages grows with increasing density.

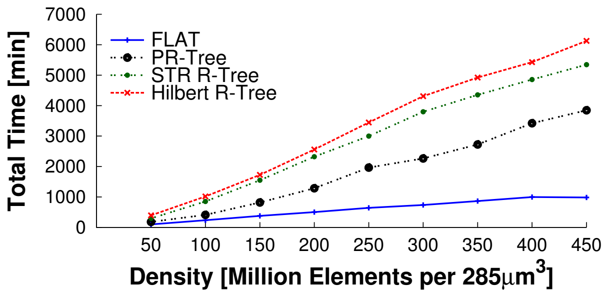


Fig. 13: Execution time for executing the SN benchmark.

More importantly however, the ratio between non-leaf to leaf page reads increases. While for the sparsest data set (50 million elements) the ratio is 2, it grows to 2.8 for the densest data set (450 million elements). This means that with increasing density, more non-leaf pages need to be read per leaf page read. This clearly shows that overlap is a problem for data sets of increasing density.

In the case of FLAT, page reads are due to the query in the seed tree as well as reading metadata and object pages. The first largely depends on the height of the tree, and hence can be considered constant, whereas the latter two depend on the result size. Because we keep the query size constant but increase the density, the result sets become bigger. This can be seen in Figure 14 (left) where we measure the number of pages read from the seed tree, from metadata, and object pages for data sets of increasing density. The page reads due to the seed tree remain constant while the ones for metadata and object pages grow with the result set size.

Because the result set grows with increasing data density, the initial cost of the seed query is amortized over a larger result set and hence the total cost per result element decreases for FLAT. For the R-Tree variants on the other hand, the pages read per result element increase because the density increases with the data set size. This increases the overlap in the R-Trees and results in a growing number of page reads per result element. The experiment in Figure 15, where we measure the pages read per result element for both approaches, clearly demonstrates this.

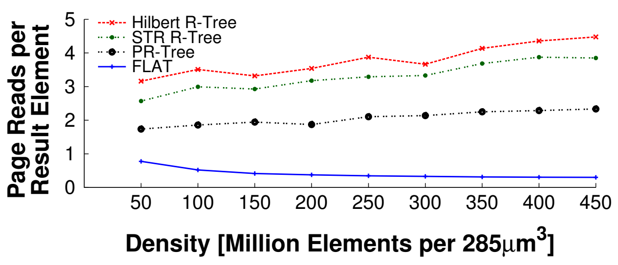


Fig. 15: Pages read per result element for the SN benchmark.

LSS benchmark results

Time & Page Read Comparison

Although overlap in the R-Tree variants does not affect big queries as much as small queries (because the overlap overhead is amortized over a bigger result set) we still expect FLAT to perform better because it does not have to retrieve hierarchical information (the subtree of non-leaf pages which connect the root with the leaf pages). The experiment measuring the page reads for executing the LSS benchmark shown in Figure 16 confirms that FLAT needs to read fewer pages to compute the result, and hence scales better for dense data sets. The relative performance among the R-Tree variants is the same as for the SN benchmark (and conforms with previous results in literature).

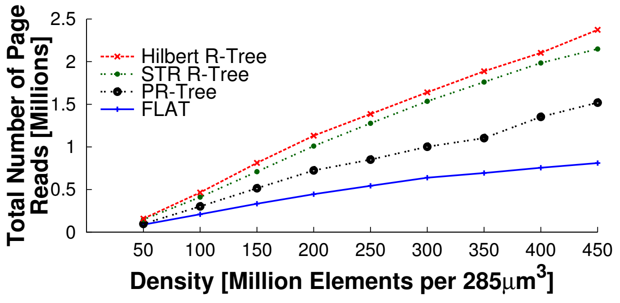


Fig. 16: Total number of page reads when executing the LSS benchmark.

Similar to the SN benchmark, the query execution time is also dominated by the page reads. For data sets of increasing density, the trend of the execution time experiment shown in Figure 17 has the same shape as the page read experiment shown in Figure 16.

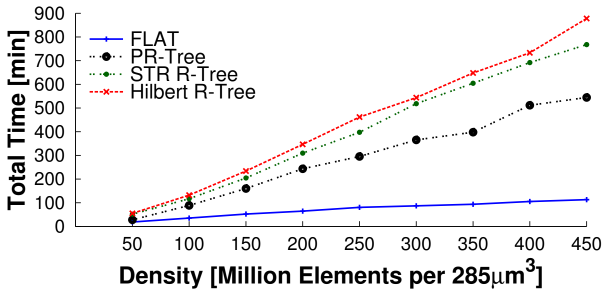


Fig. 17: Execution time for executing the LSS benchmark.

Overhead Analysis

The difference between FLAT and the R-Tree variants, however, is not as significant as for the SN benchmark. This is because the overlap only has a minor impact on larger volume queries. This can be seen in Figure 18 (right) where we analyze the ratio of retrieved leaf and non-leaf pages for the LSS benchmark for FLAT and the PR-Tree, the best R-Tree variant for this use case. For both approaches the share of object pages or leaf pages respectively is substantially higher.

The overhead for the R-Tree variants (non-leaf pages), however, is still substantially higher than for FLAT (seed tree and metadata). Instead of retrieving the subtree leading to the

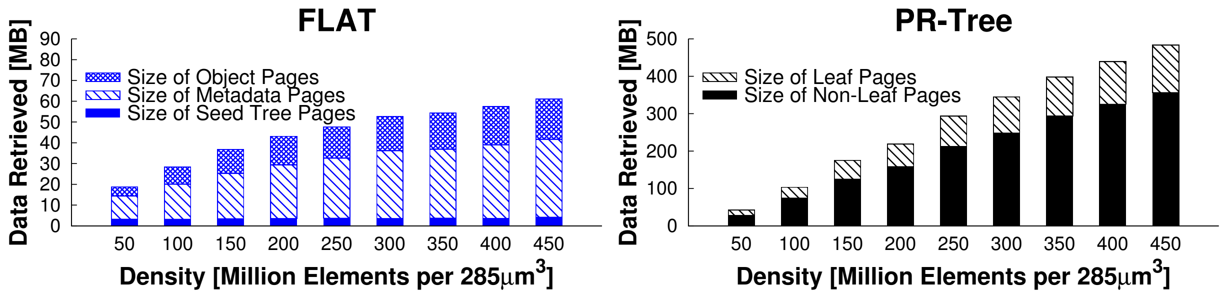


Fig. 14: Breakdown of data retrieved for both approaches for the SN benchmark.

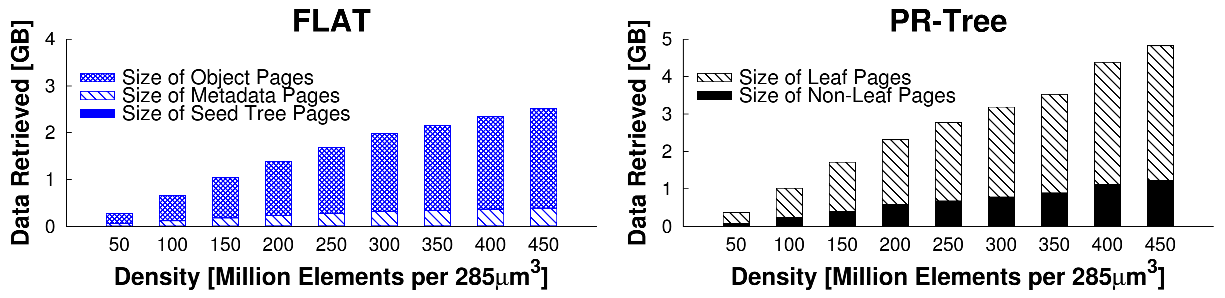


Fig. 18: Breakdown of data retrieved for both approaches when executing the LSS becnhmark.

result stored in the leaf pages, FLAT only uses the seed tree to find an arbitrary element in the range, and then uses the metadata to find the relevant object pages. It therefore scales with the size of the result set.

The difference in overhead between FLAT and the PR-Tree is significant. Comparing Figure 18 (left) with Figure 18 (right) shows that the PR-Tree has an increasing overhead with growing density (up to three times more overhead for the densest data set). What is more important, however, is how the overhead develops relative to the result set size with increasing density. The page reads per result as a function of increasing data density shown in Figure 19 illustrate this. Because FLAT amortizes the fixed cost of the seeding phase over an increasing result set, the page reads per element decrease. The R-Tree, on the other hand, retrieves a subtree of increasing size with an increasing result set size, and the page reads per element hence grow. The impact of a subtree of increasing size however is not as severe as the effect of overheads caused by the overlap in the R-Tree variants as seen in the SN benchmark. Still, with an increasing density of the data set, FLAT scales better by requiring less page reads per element than the R-Tree variants.

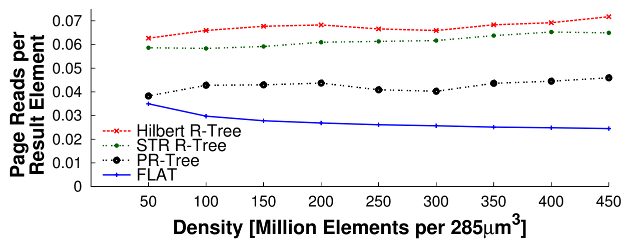


Fig. 19: Pages read per result element when executing LSS benchmark.

E. *FLAT Analysis*

The characteristics of the data set have a direct influence on the number of pointers FLAT adds to the data set which in turn has an impact on the performance of the FLAT query execution. In the following section we study what particular data set characteristics have an impact on the number of pointers, and we analyze the FLAT querying algorithm in more detail.

1) Impact of Data Set Characteristics: The number of pointers FLAT adds to the data set has an impact on the number of disk pages read to evaluate a query. The size of the metadata grows with the number of pointers as does the average number of pointers per metadata record (the number of records stays the same because it depends solely on the number of spatial elements). Both these effects lead to more metadata records read during each step of the crawling phase. In the following we therefore study the impact of different factors on the number of pointers.

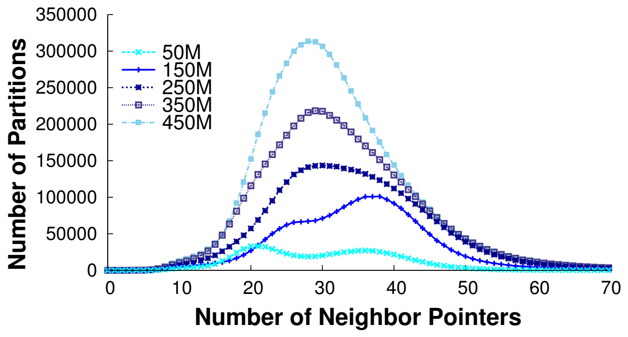


Fig. 20: Distribution of the number of pointers per partition with increasing data density.

Impact of Density
We study the impact of an increasing density of the data set on the number of pointers of each partition. For this we index neuroscience data sets of increasing density with FLAT and measure how many partitions have a given number of pointers. As the distribution in Figure 20 shows, the median stays the same and it becomes more explicit with increasing density, and appears to converge at 30. From this we infer that even if the model of the brain becomes more detailed, the average number of pointers remains constant, and with this the size of the metadata grows only linearly.

Impact of Partition Size
Because the number of pointers is determined by how many partitions overlap with each other, the major factor influencing the average number of neighbor pointers is the size of the partitions. To demonstrate this, we generate artificial data sets with 10 million elements which are uniformly randomly distributed in a volume of $8mm^3$. We calculate the partitions for the data set and then incrementally increase the size of the partitions. Figure 21 confirms that with an increasing partition size the average number of neighbor pointers grows as well.

The volumes and the aspect ratios of the elements in turn have a direct impact on the partition size because the partition needs to be stretched to accommodate the elements in it. We have designed two experiments to illustrate this.

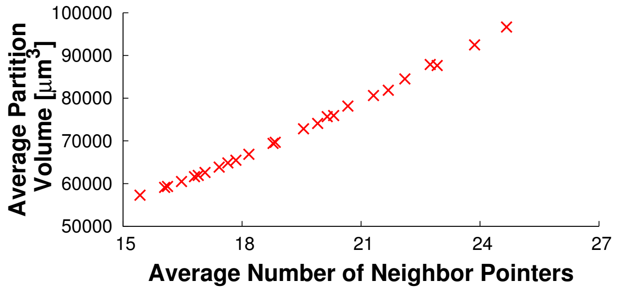


Fig. 21: Distribution of the number of pointers per partition with increasing partition volume.

In the first experiment we generate an artificial data set of 10 million uniformly randomly distributed elements in $8mm^3$, and increase the volume of each element (the length of the elements are the same along each axis) but its position remains the same. This experiment confirms that an increase of the element volume does indeed have an impact on the average number of pointers per partition, as increasing the object size by a factor of 5 incurs a 10% increase in pointers.

In the second experiment we generate a data set of 10 million uniformly randomly distributed elements each with a constant volume of $18\mu m^3$ in $8mm^3$. We keep the position the same but vary the aspect ratio as follows: for each element, its length in each dimension is randomly set between 5 and $35\mu m$. The lengths on all axes are normalized (by choosing an axis at random) in order to obtain elements of equal volume. This experiment also confirms that the aspect ratio has an influence on the number of pointers: the average number increases linearly from 17.4 to 22.9 throughout the entire range of possible aspect ratios.

2) FLAT Memory & Computation Overhead: The overhead in terms of memory footprint and computation during query evaluation is not significant. The memory footprint for FLAT increases during the crawling phase but only grows linear with the size of the result set of the query. The space used to store the book keeping information (the queues of the breadth first algorithm) remains at 0.9% of the size of the result set.

Most of the time evaluating queries is spent in retrieving data from disk. The share of time used for disk operations ranges for both benchmarks between 97.8% (for LSS on 50 million) and 98.8% (for LSS on 450 million). The remaining time, the computational overhead, is spent on processing the page data. The majority of this time is used to compute intersections of the page and partition MBRs with queries. The number of MBR intersections depends on the number of spatial elements stored on the object pages and the number of neighbor pointers for each metadata entry. As we showed in the previous subsection, the average number of neighborhood pointers converges with an increasing data set density, and hence the share of time needed for calculating MBR intersections remains constant for increasingly dense data sets.

VIII. FLAT on Other Data Sets

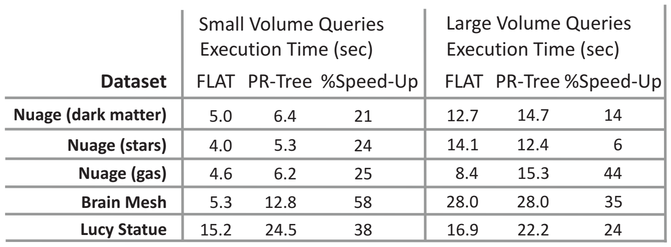
We also use FLAT to index other scientific data sets representing spatial information in order to see how it performs compared to the PR-Tree. We have taken the Nuage [15] data sets which model the n-body problem, a simulation of how the universe evolved since the big bang. These data sets contain spatial information modeled with vertices representing dark matter, gas and stars. They contain 16.8, 16.8 and 12.4 million vertices in space amounting to a size of 768, 768, 568MB respectively. In addition to this, we index a data set representing a small section of the brain containing 1600 neurons. This model is different than the other data sets used because it is modeled with a surface mesh containing 173 million 3D triangles (7.9 GB on disk). Finally we index a 3D model of the "Lucy" angel statue represented by 252 million surface mesh triangles (11GB on disk).

Like the table in Figure 22 shows, FLAT requires more time to index and also more space to store the same information for all the data sets. Similarly to the experiments with the neuroscience data sets discussed previously, the additional time and space required is only modest however.

Dataset	Index Size(MB)		Building Time (sec)	
	FLAT	PR-Tree	FLAT	PR-Tree
Nuage (dark matter)	1050	998	135	916
Nuage (stars)	1050	998	138	1021
Nuage (gas)	780	739	102	721
Brain Mesh	10939	10304	1736	9901
Lucy Statue	15558	15032	2954	21868

Fig. 22: Index size and building time for each of the data sets.

Because these models are not as densely packed as the neuroscience data sets, we do no expect FLAT to outperform the PR-Tree substantially. We benchmark with two sets of queries: the “small volume queries” set which contains 200 queries with a fixed volume of $5 \times 10^{-7}\%$ of the particular data set volume, and the “large volume queries” set which contains 200 queries with a fixed volume of $5 \times 10^{-4}\%$ of the data set volume. The location and aspect ratio of all queries is selected randomly.



	Small Volume Queries Execution Time (sec)			Large Volume Queries Execution Time (sec)		
Dataset	FLAT	PR-Tree	%Speed-Up	FLAT	PR-Tree	%Speed-Up
Nuage (dark matter)	5.0	6.4	21	12.7	14.7	14
Nuage (stars)	4.0	5.3	24	14.1	12.4	6
Nuage (gas)	4.6	6.2	25	8.4	15.3	44
Brain Mesh	5.3	12.8	58	28.0	28.0	35
Lucy Statue	15.2	24.5	38	16.9	22.2	24

Fig. 23: Execution time and speedup of small and large volume queries.

The results in the table in Figure 23 show a speed up of query execution time by using FLAT from 21-58% for “small volume queries” and around 6-44% for “large volume queries”. As discussed before, queries with larger volumes do not suffer as much from the overlap in the R-Tree as do small queries, and hence less speed up is achieved.

With these experiments we show that FLAT can also be used to improve performance on other spatial data sets. The biggest performance gains are obtained when using it on dense data sets, e.g., meshes.

IX. Conclusions

In this paper we identify the problem of indexing dense spatial data used in brain simulations. As we show, known approaches do not scale with the density of the data or require connectivity information in the data set. For dense spatial data which does not contain enough or no connectivity information, we have developed FLAT. Compared to the bulkloaded R-Tree variants, FLAT requires more space and time to index our data sets. Indexing, however, is done only rarely, when the model of the brain is updated.

More importantly however, our approach is very efficient for range queries, the predominant type of query executed by the neuroscientists working with this data. The query runtime only depends on the height of the seed index and the size of the result set. With it, queries can be answered with significantly fewer disk page reads and hence also in much less time. For a data set of only 450 million elements, FLAT already outperforms the best bulkloaded R-Tree in our experiments, the PR-Tree, up to a factor of eight for the structural neighborhood use case. The trends indicate that the benefit will be even bigger for larger and denser data sets used in the near future in the Blue Brain project.

Because we only make a few assumptions about data set characteristics, FLAT can also be used for other spatial data sets which only require infrequent updates. We demonstrate this with experiments on other scientific data sets where FLAT also reduces the query execution time.

References

[1] 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: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 347–358, 2004.

[2] R. Bayer. Binary B-Trees for Virtual Memory. In *Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control*, 1971.

[3] 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.

[4] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9):509–517, 1975.

[5] J. L. Bentley and J. H. Friedman. Data Structures for Range Searching. *ACM Computing Surveys*, 11(4):397–409, 1979.

[6] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2), 1998.

[7] Y. J. García, M. A. López, and S. T. Leutenegger. A Greedy Algorithm for Bulk Loading R-trees. In *GIS ’96: Proceedings of the 6th ACM Geographical Information Systems Conference*, pages 163–164.

[8] J. Gray, A. Szalay, A. Thakar, P. Kunszt, C. Stoughton, D. Slutz, and J. Vandenberg. Data Mining the SDSS SkyServer Database. In *Technical Report, MSR-TR-2002-01, Microsoft Research*, 2002.

[9] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD ’84: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[10] M. Hadjieleftheriou, May 2010. `http://www2.research.att.com/~marioh/spatialindex/`.

[11] C. L. Jackins and S. L. Tanimoto. Oct-trees and Their Use in Representing Three-dimensional Objects. *Computer Graphics and Image Processing*, 14(3), 1980.

[12] I. Kamel and C. Faloutsos. Hilbert R-Tree: An Improved R-Tree using Fractals. In *VLDB ’94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 500–509, 1994.

[13] D. Komatitsch, S. Tsuboi, C. Ji, and J. Tromp. A 14.6 Billion Degrees of Freedom, 5 Teraflops, 2.5 Terabyte Earthquake Simulation on the Earth Simulator. In *SC ’03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, 2003.

[14] 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.

[15] Y. Kwon, D. Nunley, J. Gardner, M. Balazinska, B. Howe, and S. Loebman. Scalable clustering algorithm for n-body simulations in a shared-nothing cluster. In *Scientific and Statistical Database Management*, volume 6187 of *Lecture Notes in Computer Science*, pages 132–150. 2010.

[16] S. Leutenegger, M. Lopez, and J. Edgington. STR: a Simple and Efficient Algorithm for R-Tree Packing. In *Proceedings of the 13th International Conference on Data Engineering*, pages 497–506, Apr 1997.

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

[18] Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical Report Ottawa, Ontario, Canada, IBM, 1966.

[19] 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: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 551–562, 2006.

[20] O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter. Bkd-Tree: A Dynamic Scalable kd-Tree. In *Advances in Spatial and Temporal Databases*, volume 2750 of *Lecture Notes in Computer Science*, 2003.

[21] H. Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

[22] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *VLDB ’87: Proceedings of the 13th International Conference on Very Large Data Bases*.