Spatial Data Management Challenges in the Simulation Sciences

Thomas Heinis, Farhan Tauheed, Anastasia Ailamaki
Data-Intensive Applications and Systems Laboratory, École Polytechnique Fédérale de Lausanne, Switzerland
{thomas.heinis,farhan.tauheed,anastasia.ailamaki}@epfl.ch

ABSTRACT

Scientists in many disciplines have progressively been using simulations to better understand the natural systems they study. Faster hardware, as well as increasingly precise instruments, allow the construction and simulation of progressively advanced models of various systems.

Governed by algorithms and equations, the spatial models at the core of simulations are changed and updated at every simulation step through spatial queries, implementing massive updates. Therefore, the efficient execution of these numerous spatial queries is essential.

Two reasons render current spatial indexes inadequate for simulation applications. First, to ensure quick access to data, most of the spatial models in simulations are stored in memory. Most spatial access methods, however, have been optimized for use on disk and are not efficient in memory. Second, in every time step of a simulation, almost all spatial elements change their position, challenging update mechanisms for spatial indexes.

In this paper we discuss how these challenges create opportunities for exciting data management research.

1. INTRODUCTION

Many scientists today across different disciplines no longer solely study a phenomena in vitro or in natura. Instead, to better understand the phenomena, they simulate it in large-scale computing clusters or supercomputers. Simulating the phenomena allows them to develop a better understanding of it by testing and refining their hypothesis through a cycle of building a spatial model, simulating it, analyzing its output, refining the model, and finally simulating it again.

Increasingly precise instruments, high-throughput analysis technology (e.g., shotgun proteomics), abundance of computational power and an ever better understanding of the phenomena, enables the development of increasingly detailed spatial models. Neuroscientists involved in the Blue Brain project (BBP) [19], for example, have started to simulate parts of the rat brain at the molecular level and will soon move to the subcellular level (e.g., modeling the neurotransmitter). Material scientists in the computational solid mechanics lab (LSMS) and the Swiss Super Computing Center (CSCS) are currently building and simulating progressively detailed models of material deformation and meteorological phenomena. Similarly, other fine-grained models across various disciplines are being developed, for example, the human arterial tree [9] in computational fluid dynamics research, earthquakes [1] in geology and protein synthesis [25] in computational biology research.

During the simulation, the spatial models located in the main memory of the simulation infrastructure are accessed and updated using spatial queries for two crucial applications. First, spatial queries are needed to compute the model at each simulation step, for example, in n-body simulations in physical cosmology [5] the position of each celestial object at time step t_{i+1} has to be computed based on the gravitational field (and thus the locations) of its neighbors at time step t_i. Second, the running simulation needs to be analyzed at runtime, for purposes of statistical analysis, visualization etc., and consequently spatial queries need to be executed at every time step to retrieve a subset of the model.

For both applications, updating the model and monitoring the simulation, a vast number of spatial queries need to be executed at every step of the simulation. Known spatial indexes can significantly help to speed up queries on the models, yet two factors render state-of-the-art indexes inadequate:

Spatial Indexes In Main Memory: Spatial models at the core of simulations are stored in the main memory of the simulation infrastructure. Current spatial indexes, however, have been optimized for use on disk and minimize disk access because data transfer dominates query execution time in disk-based indexes. In memory, however, the time for data transfer is comparatively small and, to speed up query execution, computation needs to be reduced.

Massive Changes: Numerous simulations change the entire spatial model at every time step. Although the location change of each element is minimal for most elements, almost all of them change position. This rate and magnitude of change challenges current spatial indexes' update mechanisms, which are designed for only small (position and number of elements) or predictable changes. New data structures supporting efficient large-scale updates need to be designed for the simulation sciences.

Both factors require a redesign of spatial indexes for the simulation sciences, therefore the remainder of this paper discusses the research challenges of simulation scientists iden-

tified in our work. First, further background on simulation applications is given and then the challenges are discussed in detail by illustrating them with concrete examples. We demonstrate the shortcomings of today's solutions and we sketch possible research directions. Although the two challenges are analysed separately, they are not mutually exclusive and we will ultimately have to develop indexing approaches to address both challenges in unison.

2. SPATIAL DATA IN SIMULATIONS

Simulations have become a standard tool for the detailed study of a natural phenomena by domain scientists. In the following, we first give an overview of simulations and then discuss the central role that spatial data and queries play.

2.1 Simulation Background

Computer simulations are typically used to study the behavior of systems so complex that they can no longer be solved analytically. Given a model and an initial state, simulations calculate and approximate the subsequent states of the model in discrete time steps (see Figure 1). The difference of state between two time steps is calculated based on the two core parts of a model, namely a) the elements (each of which has a state) and b) the interactions between elements. For example, in an n-body simulation from cosmology the celestial bodies are the elements whose gravitational fields are the interactions between them. The interactions are typically modeled with differential equations that cannot be solved analytically and hence have to be numerically approximated during simulation.

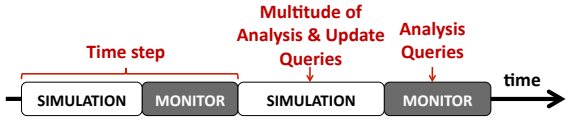

Figure 1: Timeline of a time-stepped simulation.

The state of a simulation is usually kept in the memory of the simulation infrastructure because accessing the disk is too slow. Efficient data structures required to access and to query the state in memory for monitoring (visualization, online validation etc.) and updating the model (computing the state of the next time step) are consequently key. Given the high number of queries and their repeated execution in every time step, optimizing the execution time of the queries is crucial to speed up the entire simulation. As Figure 1 shows, during the simulation phase analysis/update queries are executed to update the model and during the monitoring phase analysis queries are executed to monitor the progress of the simulation.

2.2 Spatial Queries

In many simulations like n-body simulations in cosmology, earthquake simulations and others the model at the core is spatial, i.e., the elements of the models have spatial attributes (a position). The thousands of queries used to update and analyze the model at every step of the simulation consequently are spatial queries executed on the model stored in main memory. In the following we discuss the most important spatial queries used in the context of simulations.

Range Queries: Executing spatial range queries on the model during simulation is crucial for several applications: e.g., for the local analysis of tissue density in neuroscience models and the analysis of deforming alloy in material sciences. The most important application that needs to execute range queries is the in-situ visualization of the progressing simulation. For visualizations, as well as analyses, thousands of range queries need to be executed between two simulation steps at locations that cannot be anticipated.

Nearest Neighbor Queries: Computing the nearest neighbors of multiple points on the model is crucial in several use cases. Material scientists, for example, need nearest neighbor queries to simulate material deformation [2]: the position of a vertex in the discretized material model at the next simulation step is computed based on the force fields of its nearest neighbors. Neuroscientists similarly need to determine the nearest neighboring neurons of a particular neuron to simulate its deformation and to compute its shape (and thus to build bio-realistic models).

Spatial Joins: In many simulations, intersection of the elements in spatial models does not realistically reflect the system modeled (celestial bodies, for example, cannot intersect in reality). To detect intersections, the entire model needs to be spatially joined with itself at every simulation step. In other simulations, determining the proximity between elements (a distance or a spatial join) is an integral part of the simulation result. Neuroscientists simulating the co-growth of neurons, for example, need to perform a spatial join to determine the location of synapses: wherever two neurons are within a given distance of each other, they will form a synapse to communicate with each other [17].

3. IN-MEMORY SPATIAL INDEXING

Simulation hardware, i.e., supercomputers, clusters, cloud deployments, is becoming ever more powerful, both, in terms of main memory capacity as well as CPU. The increasing main memory in particular encourages scientists to build and simulate bigger spatial models. With bigger spatial models, however, their organization in memory is crucial for the efficient execution of spatial queries.

3.1 In-Memory Challenges

Most of today's spatial indexes are optimized to retrieve as little data as possible from the index. Doing so for disk-based indexes is crucial because the vast majority of time is spent on retrieving data from disk as an experiment shows. In this experiment we index a dataset of 200 million spatial elements with an R-Tree and execute 200 queries with a selectivity of 5×10^{-4}% at random locations. As the result in Figure 2 shows, 96.7% of the total time is spent on reading data from disk in case of the disk-based R-Tree. Past research on disk-based spatial indexes has therefore primarily focused on reducing the number of pages read from disk.

Executing the same experiment in memory, the execution time drops from 2253 seconds on disk to a mere 40 seconds. More importantly, however, is the shift of cost demonstrating the optimization potential: in memory, as little as 3.3% of the overall time is spent on reading data while computations (intersection tests, following pointers etc.) take the vast majority, i.e., 95.3% of the overall time. Clearly, reducing the data read still helps to speed up query execution. The far bigger potential, however, lies in optimizing the computation.

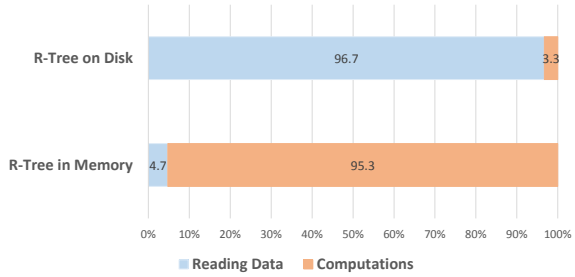

Figure 2: Query execution time breakdown of the R-Tree in memory and on disk.

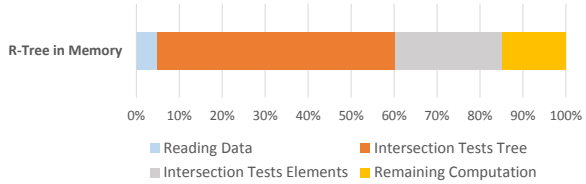

Figure 3: Query execution breakdown of the R-Tree in memory.

Breaking down the time spent on computations further shows the proportion of time spent on intersection tests. Intersection tests are necessary to (a) test if elements indeed intersect with the query range and (b) to navigate through the tree structure of the R-Tree. As Figure 3 shows, most of the time (around 80%) is spent on performing intersection tests. 55% of the time are spent on intersection tests in the tree structure of the R-Tree.

The large share of intersection tests with inner nodes of the tree demonstrates the overhead of using tree structures to access the data. Overlap of inner nodes clearly leads to an increase in the number of intersection tests, but even without it, the overhead of intersection tests is considerable and this small scale experiment (simulations use substantially bigger datasets) shows the limitations of approaches that need tree structures to find elements.

The share of intersection computations testing the intersection of single elements with the query, however, is also considerable with 25% of the overall time. Clearly, the number of intersection tests needs to be minimized.

The exact experimental setup can be found in Section A.

3.2 State of the Art

Most spatial indexing approaches developed in the last decades have been developed for disk. As a consequence, all their data structures are aligned for the disk page size and are designed to minimize the data read from disk. Arguably the most seminal data structure developed for disk is the R-Tree [10] which, by using a tree structure to access the spatial data, supports the execution of a broad range of spatial queries (range query, kNN query, spatial join etc.).

The R-Tree, however, also has inherent problems that considerably degrade its performance, namely overlap of the inner nodes of the tree structure and dead space [28]. Numerous extensions (Priority R-Tree, R*-Tree, R+-Tree, etc. see survey [8]) reduce the overlap and hence improve performance, but the fundamental problem of overlap remains [28].

Because of its popularity, the R-Tree is one of the few disk-based spatial indexes which has also been optimized for memory. The resulting CR-Tree [16]) optimizes the R-Tree for use in memory by making the nodes fit into a multiple of the cache block through compression, pointer reduction and quantization of the bounding boxes. Optimizing it for memory, however, only speeds up query execution by a factor of two over the R-Tree as experiments [16] show because the fundamental problem of overlap remains unaddressed.

Other spatial indexing approaches used in memory are point access methods like the KD-Tree [4], the Quadtree [24] and the Octree [14]. Supporting volumetric objects with these indexes can be accomplished by replicating elements which occupy several partitions on the leaf level. However, by doing so, the index size is increased massively. Other extensions avoid replication by increasing the size of the partitions (e.g., loose Octree). Bigger partitions for space-oriented approaches, however, introduce substantial overlap and therefore increase unnecessary child traversals (and comparisons) similar to the R-Tree.

Several approaches have been conceived for joining spatial datasets on disk [15]. Most of them can also be used in memory but will perform suboptimal because they are designed to minimize disk access but not the number of comparisons (the major bulk of work for in-memory spatial joins [21]). Until recently only the nested loop join and the sweep line were specifically designed for use in memory. Judging that neither is efficient, we designed TOUCH [21], which outperforms both as well as disk-based approaches used in memory.

3.3 Research Directions

To develop new in-memory spatial indexes for the simulation sciences (and for spatial datasets in memory in general), we have to rethink indexes and build on key insights drawn from previous work and ideas developed in this paper.

First, using disk-based approaches to index and query spatial data in memory leads to an inefficient use of memory caches and hence degraded performance. Indexes used in memory must be optimized for memory hierarchies by making the size of their nodes a multiple of the cache block size [12]. Node sizes substantially smaller than used on disk (on disk sizes 4KB or bigger are typically used) achieve good performance (between 640Bytes and 1KB [31]). In this sense the CR-Tree [16] is a step in the right direction. Optimizing the size of the data structures, however, will not improve performance considerably because the data transfer is only a small fraction of the overall time.

Second, and most importantly, as the experiment in Figure 3 shows, a considerable share of the total time is spent on traversing the tree. Hierarchically organized tree structures needed to find elements should thus be avoided. Organizing the data as a tree is only needed if the data is partitioned non uniformly, i.e., using data-oriented partitioning (R-Tree) or non-uniform space-oriented partitioning (Octree).

Third, data-oriented partitioning for the execution of range queries in memory is unnecessary. Partitions resulting from data-oriented partitioning can be very big (extend massively in one or several dimensions) while still grouping spatially neighboring elements. As Figure 4 shows, a range query intersecting with such a partition may contain only few of the partition's elements, yet all elements need to be tested for intersection [13], leading to unnecessary intersection tests. This degrades performance particularly in memory where

intersection tests account for the majority of time.

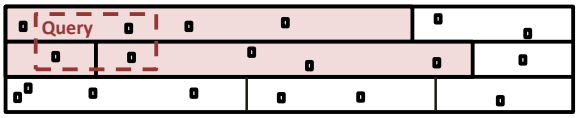

Figure 4: A range query intersecting with narrow partitions (shaded) leads to unnecessary tests.

Data-oriented partitioning is an artifact of disk-based indexes: by using non-uniform partitioning in all dimensions, one can ensure that each partition (and also inner nodes) fits on exactly one disk page. The latter ensures that disk pages are almost entirely filled and no disk bandwidth is wasted retrieving partially empty pages. In memory, however, there is no abstraction of a page size, only the much smaller (typically 64Bytes) cache line that gives considerably more flexibility (underfilling a cache line incurs a much smaller penalty). Data-oriented partitioning in memory is hence only useful to avoid replication of elements.

Fourth and finally, potential improvements like compression as proposed for the CR-Tree [16] will not significantly improve performance: the data transfer is only a small fraction of the overall time and reducing data size will not give a major improvement of the total time. Also, the gain for reading less data is unlikely to offset the additional overhead required for decompression [13].

One direction to develop novel spatial indexes for main memory may be to use a single uniform grid and therefore to avoid the tree structure needed for access [26]. Choosing the proper resolution, however, is difficult: a too coarse grained grid means that too many elements need to be tested for intersection. This is a particular problem for kNN queries where all elements of (potentially several) partitions need to be tested to find the k nearest elements.

Clearly, the optimal resolution depends on the distribution of location and size of the spatial elements and an analytical model needs to be developed to determine it for a given dataset. The optimal resolution, however, also depends on the size of the queries which cannot be known a priori. A solution to the resolution challenge may thus be to use several uniform grids each with a different resolution: queries may be split and each part (or the whole query) is executed on the grid with the best suited resolution.

A possible approach for kNN queries could be to use locality sensitive hashing (LSH, e.g., [3]). LSH has traditionally been used for similarity search in very high dimensions but can potentially also be used for finding nearest neighbors in low dimensions. Crucially, LSH avoids a tree structure to organize the data and instead uses several (spatial) hash functions to index each spatial element. Difficult to implement efficiently on disk, LSH's hash buckets can also easily be optimized for use in memory by making them cache-aware, i.e., by aligning the hash buckets for a multiple of the cache line.

Recent work [21] for the spatial join in memory considerably reduces the number of comparisons needed and therefore speeds up the process. The proposed method, however, depends on a costly data-oriented partitioning & indexing step prior to the join. An approach based on a grid (similar to PBSM [15]) optimized for memory may not necessarily speed up the join, but will certainly speed up the preprocessing/indexing and thus the overall join.

4. MASSIVE UPDATES

In many simulations the entire spatial model undergoes massive changes in each step. The changes are massive in that they affect a vast majority of the elements, but most elements only move minimally. In neural plasticity simulations, for example, all elements change position in every step of the simulation, yet each element only shifts minimally.

4.1 Massive Update Challenges

The majority of spatial indexes today support the execution of updates efficiently. In case the entire dataset changes, however, update mechanisms are no longer efficient and it is often cheaper to rebuild the entire index [27]. The rate of change may challenge the use of indexes in this scenario altogether. Depending on how many queries are executed, rebuilding an index may no longer pay off as the cost cannot be amortized over enough queries and using no index, i.e., a linear scan over the dataset, may be faster.

The changes in an neural plasticity simulation, for example, are massive yet minimal as an experiment shows. In each of the one thousand simulation steps in a sample run of a neural simulation, all elements move, but only by $0.04\mu m$ (in a universe with volume of $285\mu m^3$) on average with less than 0.5% of elements moving more than $0.1\mu m$. Updating all elements of this application in an R-Tree takes 130 seconds at every simulation step. Building the new R-Tree index from scratch, on the other hand, only takes 48 seconds. For this experiment updating only is faster than a rebuild if less than 38% of the dataset change in a time step. The experimental setup is described in Section A.

More efficient update mechanisms particularly devised for quick changing data (moving data) do not considerably improve performance of the updates. One class of approaches assumes predictability of the movement of the elements. In simulations the movement patterns, however, are unpredictable and this class of approaches cannot be used. Other moving object indexes avoid single updates by either batching them or by indexing them with looser bounding boxes. While these indexes indeed reduce maintenance overhead, overhead is shifted to query execution: in case of batched and buffered updates, the buffer and the index have to be searched for every query and in case of loose bounding boxes, every element has to be checked to see if it is indeed in the query. Completely rebuilding indexes quickly becomes more efficient [27] than these update mechanisms as well.

For executing range queries (or similar spatial queries) the linear scan can be very fast, depending on the number of queries asked and in case many query can be batched together. The spatial join, on the other hand, always depends on an index or similar data structure as the naive method is not linear in the number of elements in the datasets, but quadratic (nested loop join). Maintaining a data structure supporting the spatial join will thus almost always pay off.

4.2 State of the Art

Although the majority of past approaches have been primarily designed for disk, they can also be used in memory and we consequently discuss all approaches.

Update strategies have been devised for almost all spatial indexes. For the R-Tree, for example, several strategies have been devised (through reinsertion of elements like the

R*-Tree [8] or with a bottom up approach [26]). If, however, a majority of the objects need to be updated, rebuilding or bulkloading the index is an efficient alternative to updates [7, 27]. Efficient bulkloading methods have been developed for many spatial indexes like the Octree, the KD-Tree [4] or memory optimized R-Trees [16]. Several bulkloading methods (see survey [8]) have been devised for the latter.

Update strategies to perform massive numbers of updates for moving objects have also been devised [20]. A first class assumes that moving objects have a predictable trajectory, i.e., approximately constant speed and direction, and this class thus only indexes the trajectory (STRIPES, TPR*-Tree, TPR-Tree, see survey [20]). Updates are only needed if speed or trajectory change. These approaches do not work well for simulations because the movement of objects cannot be predicted. The movement of objects is ultimately what the simulation determines.

A second class does not assume predictability, but introduces a grace window: instead of using a tight bounding box, objects a packed in a looser grace window. With this, the index does not have to be updated if a object only moves in the grace window [18, 30], thereby reducing the number of updates. Still updates are required frequently and, by introducing an imprecision in the index structure, the burden is shifted to the query execution where objects need to be tested for intersection with the query [7, 27].

Buffering the updates to reduce operations on the index [6] similarly shifts the burden to query execution: when computing the query result, buffer and index need to be checked, thereby increasing the overhead.

Performing a complete linear scan over the dataset is the most basic approach to compute the result of a range query. While it has no memory overhead, query execution time will not scale as it directly depends on the dataset size.

4.3 Research Directions

As we discussed previously, update mechanisms that need more operations than there are elements are unlikely to outperform a simple linear scan. Key to any method to execute massive updates efficiently therefore is to avoid updating all indexed elements.

A first research direction is to use indexes that predominantly depend on the dataset itself for query execution. The dataset is updated by the simulation application anyway and is always up to date. If an index uses the dataset directly, then it does not need to perform any updates. Similar ideas have already been explored. DLS [22] uses an approximate index as well as the mesh connectivity to execute range queries: the approximate index (which only needs to be updated infrequently) is used to find a start point near the query range and the mesh connectivity is used to a) find the query range and b) to find all results in the range. DLS, however, only works for convex meshes (without holes).

OCTOPUS [29] takes the DLS ideas into memory but also supports concave meshes. To ensure that query execution still retrieves the entire range query result in face of concave meshes, OCTOPUS takes as start point several elements on the surface. For datasets other than meshes, disk-based FLAT [28] adds connectivity (neighborhood) information to the dataset and then uses it to execute spatial queries (similar to DLS or OCTOPUS). The same idea can potentially also be used in memory.

A second direction is to exploit that the movements of most elements are small. Although exploiting this characteristic of the use case does not give impressive results with R-Trees (QU-Trade, LUR-Tree), using grids will considerably lower the overhead of updates. Clearly the small movement means that only few elements switch grid cell in every step, thereby requiring few updates to the data structure.

To perform the spatial join efficiently on datasets with massive changes, the join requires some form of index structure. Not using any index structure results in a nested loop join with n^2 comparisons. A data structure is clearly needed to avoid unnecessary comparisons of objects that are far apart from each other. The sweep line approach does not ensure that only spatially close objects are compared [21]. Any other approach based on the R-Tree or similar index (see survey [15]) suffers from update mechanisms that require substantially more operations than there are elements in the dataset.

Using grids [26, 23] where objects are quickly assigned to grid cells is an interesting research direction for the spatial join as well. Only objects in grid cells need to be compared with each other, thereby substantially reducing the comparisons. If, in addition, the size of the grid cells is chosen very small, then pairs of elements do not need to be tested for intersection: if the grid cell size is smaller than the smallest element size, then objects in the same cell intersect by definition. A grid cell size considerably smaller than the elements, however, may also lead to excessive replication. In this case, elements may not be assigned to all intersecting cells, but elements in neighboring cells need to be compared with each other to limit replication.

5. CONCLUSIONS

In this paper we identify two challenges that render today's spatial indexing approaches inadequate in the simulation sciences. The gap between the current state of the art and the need of the simulation scientists is already considerable and will keep growing.

The ultimate goal in supporting the simulation sciences is to develop a new class of spatial indexes that work efficiently in memory and can either be built easily and quickly or that efficiently support updates on a massive scale. Although we presented the two challenges separately, the solutions to them are not mutually exclusive at all. What is needed are spatial indexes for memory that support large-scale updates.

The solution, i.e., reconciling approaches that tackle either challenge, is a new point in the design space: a spatial index that executes spatial queries and the spatial join faster than without index, but at the same time is faster to update or rebuild. Indexes in this new class are unlikely to execute spatial queries faster than known spatial indexes, but their build or update cost will be substantially smaller and hence they will speed up the overall process (index building and querying). The new indexes will ultimately trade off query execution time for substantially faster index build time.

As the discussion of research directions shows, an approach to address both challenges is likely to be based on grids. As opposed to disk, grids are easy to implement efficiently in memory. As we argued, they avoid a costly tree structure and because they are based on space-oriented partitioning, they effectively reduce the number of intersection tests. At the same time, given the small positional changes of spatial elements during simulation, only few updates are needed to be performed, therefore also addressing the chal-

lenge of massive updates.

Clearly the simulation sciences face more challenges than only the two we have identified. Many challenges center around out-of-core model building, mining simulation results, optimizing data structures for new storage media (e.g., SSD) as well as challenges during simulation (e.g., in-situ compression). The two challenges we discussed, however, already offer exciting research opportunities in data management well beyond the examples we discussed. Tackling these challenges will not only advance the state-of-the-art in data management in general (other applications and domains will benefit from efficient in-memory indexes as well), but will also help to advance simulation sciences.

APPENDIX

A. EXPERIMENTAL SETUP

The experiments are run on a Linux Ubuntu 2.6 machine with 2 quad CPUs AMD Opteron, 64-bit @ 2700MHz, 4GB RAM and 4 SAS disks (300GB each) striped to 1TB. We use an available implementation of the STR R-Tree [11] and set page and node size to 4K.

The dataset used contains 200 million spatial elements (resulting in 9GB on disk) in a volume of $285\mu m^3$. The data used is a neuroscience dataset representing 500'000 neurons in space (each modeled with thousands of cylinders).

All experiments are run with an initially cold cache and the cache is cleaned between any two queries. A more detailed description of the dataset (structure) and the experimental methodology can be found in [28, 29].

B. REFERENCES

[1] V. Akcelik, J. Bielak, G. Biros, et al. High Resolution Forward And Inverse Earthquake Modeling on Terascale Computers. *Supercomputing '03*.

[2] G. Anciaux, S. B. Ramisetti, and J. F. Molinari. A Finite Temperature Bridging Domain Method for MD-FE Coupling and Application to a Contact Problem. *Computer Methods in Applied Mechanics and Engineering*, 205(0):204–212, 2012.

[3] A. Andoni and P. Indyk. Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. *Communications of the ACM*, 51(1), 2008.

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

[5] E. Bertschinger. Simulations of Structure Formation in the Universe. *Annual Review of Astronomy and Astrophysics*, 36(1):599–654, 1998.

[6] L. Biveinis, S. Šaltenis, and C. S. Jensen. Main-memory Operation Buffering for efficient R-tree Updates. In *VLDB '07*.

[7] J. Dittrich, L. Blunschi, and M. A. Vaz Salles. Indexing moving objects using short-lived throwaway indexes. In *SSTD '09*.

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

[9] L. Grinberg, T. Anor, J. R. Madsen, A. Yakhot, and G. E. Karniadakis. Large-scale Simulation of the Human Arterial Tree. *Clinical and Experimental Pharmacology and Physiology*, 36(2):194–205, 2009.

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

[11] M. Hadjieleftheriou, May 2012. `http://www2.research.att.com/~marioh/spatialindex/`.

[12] R. A. Hankins and J. M. Patel. Effect of Node Size on the Performance of Cache-Conscious B+-trees. *SIGMETRICS '03*.

[13] S. Hwang, K. Kwon, S. Cha, and B. Lee. Performance Evaluation of Main-Memory R-trees. In *SSTD '09*.

[14] 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.

[15] E. H. Jacox and H. Samet. Spatial join techniques. *ACM TODS*, 32(1):7, 2007.

[16] K. Kim and K. Kwon. Optimizing Multidimensional Index Trees for Main Memory Access. *SIGMOD '01*.

[17] 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.

[18] D. Kwon, S. Lee, and S. Lee. Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree. In *MDM '02*.

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

[20] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-temporal Access Methods. *IEEE Data Engineering Bulletin*, 2003.

[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] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2), 1984.

[25] K. Y. Sanbonmatsu, S. Joseph, and C.-S. Tung. Simulating movement of tRNA into the ribosome during decoding. *Proceedings of the National Academy of Sciences*, 102(44):15854–15859, 2005.

[26] D. Sidlauskas, S. Saltenis, C. W. Christiansen, J. M. Johansen, and D. Saulys. Trees or grids?: Indexing moving objects in main memory. In *GIS '09*.

[27] B. Sowell, M. V. Salles, T. Cao, A. Demers, and J. Gehrke. An Experimental Analysis of Iterated Spatial Joins in Main Memory. In *VLDB '14*.

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

[29] F. Tauheed, T. Heinis, and A. Ailamaki. OCTOPUS: Efficient Query Execution on Mesh Data. In *ICDE '14*.

[30] K. Tzoumas, M. L. Yiu, and C. S. Jensen. Workload-aware Indexing of Continuously Moving Objects. In *VLDB '09*.

[31] R. Zhang and M. Stradling. The HV-tree: a Memory Hierarchy Aware Version Index. *VLDB '10*.