On The adoption of Spatial Indexing for Scientific Big Data Applications

André Demori[1,2], Douglas E. M. de Oliveira[1,2], João Guilherme Rittmeyer[1], Eduardo Ogasawara[3], Fabio Porto[1]

[1]Laboratório Nacional de Computação Científica (LNCC)
Extreme Data Lab (DEXL)
CEP: 22651-075 – Petrópolis – RJ – Brazil

[2]Faculdade de Educação Tecnológica do Estado do Rio de Janeiro (FAETERJ)
CEP: 22651-075 – Petrópolis – RJ – Brazil

[3]Centro Federal de Educação Tecnológica Celso Suckow da Fonseca (CEFET/RJ)
CEP: 20271-110 – Rio de Janeiro – RJ – Brazil

{demori,ericson,joaonr,fporto}@lncc.br, eogasawara@ieee.org

Abstract. *Scientific applications involve the processing of huge data sets. In the presence of spatial data, such as in astronomy, indexes may be constructed avoiding unnecessary scans of the whole data set when looking for objects in a spatial neighborhood. However, different existing indexing structures exhibit varying performance. In this paper we experimentally investigate the performance of spatial indexing structures considering their implementation in Big Data frameworks, such as Spark. We privilege, as selection criteria, the amount of memory space consumed and the elapsed-time taken to answer neighborhood queries. We consider three very different indexing choices: (a) Quad-tree, a traditional 2D indexing structure; (b)Slim-tree, a metric index and (c) PH-tree, a multidimensional binary indexing structure. We show that Slim-tree allocates orders of magnitude less memory than its competitors. This is a very important criterion for in-memory frameworks, such as Apache Spark. However, as the amount of indexed objects increases PH-tree becomes a clear winner regarding the query elapse-time. The reported analysis may shed light on strategies to extend Apache Spark to support scientific spatial applications.*

1. Introduction

Scientific applications involve the processing of huge data sets. In astronomy, for instance, the results of large surveys, such as the Sloan Digital Sky Survey (SDSS), delivers data releases openly accessible to scientists and the public in general[1]. The last SDSS data release, DR15, includes more than 1 billion sky objects, each one comprising tens of describing attributes. A common approach to explore huge scientific data sets, such as DR15, is to design and implement dataflows to run in Big Data frameworks, such as Apache Spark (Spark for short). As a general design model, Spark dataflows run on data partitioned onto a shared-nothing cluster. A Spark dataflow is made of transformation and actions. Both sequentially process the complete set of objects in a given partition. Moreover, the basic data structure processed by Spark transformations, such as Resilient

[1]http://www.sdss.org

Distributed Dataset (RDD), are kept in memory. Thus, a Spark dataflow may heavily use memory and sequentially process the complete set of objects in partitions. As it turns out, scientific applications, such as SDSS dataflows, place objects in some system of spatial coordinates and use objects spatial locality in queries to establish neighborhood relationships. Solving neighborhood queries in Spark becomes an expensive task, potentially running in quadratic time in the number of objects in a partition. Depending on the partitioning criteria, a given data partition may involve tens of thousands of objects. A traditional approach to avoid expensive scans in database literature is to use indexes. In particular, a list of existing spatial indexing structures is available for supporting indexing queries, such as a neighborhood query. In this paper, we investigate three well known in-memory data structures and their performance to index spatial objects and efficiently support neighborhood queries. We consider: (a) Quad-tree [Samet 1984], a traditional 2D indexing structure; (b)Slim-tree, a metric index [Traina et al. 2000], and (c) PH-tree, a multidimensional binary indexing structure [Zäschke et al. 2014]. We show that Slim-tree allocates orders of magnitude less memory than its competitors. This is a very important criterion for in-memory frameworks, such as Spark. However, as the amount of indexed objects increases PH-tree becomes a clear winner regarding query elapse-time. The reported analysis may shed light on strategies to extend Apache Spark to support scientific spatial applications. The remaining of this paper is organized as follows. In Section 2, we present the problem we investigate in this paper. Next in Section 3, we present the three spatial index structures investigated and the spatial range query. Section 4 presents the environment and the objective of each experiment. Next, Section 5 discusses the experiments results, followed by Section 6, where related work is described putting this work into context. Finally, Section 7 concludes.

2. Problem Formulation

We consider a context involving large scientific data sets of spatial objects. Each object spatial locality is identified by a value in a coordinate system, such as the 2D Equatorial coordinate system based on right ascention (ra) and declination (dec). Moreover, the data set may be split into data partitions (i.e. subsets of the data set, not necessarily disjoint), which are allocated to nodes in a shared-nothing cluster [Ozsu and Valduriez 2011]. Spatial data are processed by dataflows. A dataflow is composed of transformations that process objects in a in-memory data partition and produce a new data partition. Moreover, we consider transformations whose semantics involve, for each object in a partition, the computation of the set of spatially neighboring objects, under a maximum radius criterion. As an example, we may consider *The Point Pattern Search* problem [Porto et al. 2018b]. We want to experimentally investigate alternative spatial indexing structures that could be integrated to Big Data frameworks, such as Apache Spark, to boost spatial neighborhood queries. In this context, the following desired characteristics are evaluated: (a) memory usage; (b) neighboring queries elapsed-time; (c) sensitivity to data density.

3. Spatial Neighborhood Querying

In this section, we highlight the spatial neighborhood querying context. We introduce the main characteristics of the three spatial indexing structures investigated in this paper: Quad-tree, Slim-tree and PH-tree. Next, we introduce the principles behind range queries in spatial domains.

3.1. Big Data Spatial Indexes

There is a significant number of spatial indexing structure families. Our intention in this work is not to provide an extensive and complete evaluation of existing spatial indexing structures, but rather to apply a systematic approach to compare three important representatives of the family of such indexing structures. We extend the results we had presented in [Khatibi et al. 2017] and [Porto et al. 2018a] regarding the PH-tree and the Quad-tree and we additionally include in the comparison the Slim-tree metric index.

3.1.1. Slim-tree

Slim-tree [Traina et al. 2000] is a Metric Access Method (MAM). This access method works with objects in a metric space, composed of set objects and a function establishing a distance between any two objects of the set. MAMs select objects to become representatives of data subsets. From this, the distance of a new element to each representative c is calculated and arranged in its structure so that the object becomes part of the data subset of the less distant representative.

From a query point of view, Slim-tree applies triangular inequality to discard elements that are not part of the response set. In this context, the main objectives of an MAM are to reduce the number of distance calculations and the number of disk accesses in the execution of operations based on distance.

In short, the basic idea of such data structures is to choose a set of central elements and to apply a distance function to cluster the remaining objects into appropriate (with respect to distances) subsets.

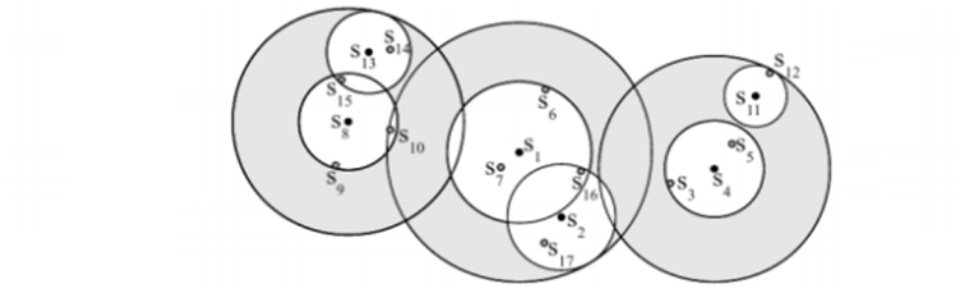

Figure 1. Slim-tree - Spatial Distribution[Traina et al. 2000]

A tree considered efficient has little overlap among its indexed entries. The Slim-tree was developed to reduce overlap between regions at each level. As the overlap increases, the efficiency of the structures decreases, since all nodes covered by a query region have to be processed during a search operation.

Slim-tree insertion happens from the root node, as in a B+ tree [Ramakrishnan 2002]. The structure is traversed until a leaf node is found. The goal is not to increase the radius of coverage. If the new object is not within the radius of coverage of any node, the node whose distance with the new element is the lowest is selected. If more than one node contains the object within its coverage radius, an algorithm *ChooseSubtree* decides on the object placement. [2]

[2]algorithm not shown in this paper.

3.1.2. PH-tree

PH-tree [Zäschke et al. 2014] - PATRICIA-Hypercube-tree belongs to the Quad-tree family, but provides much better efficiency and scalability with greater dimensionality support. It is an unbalanced structure that basically builds on the quad-tree and divides the space on each node into several dimensions. This reduces the number of nodes in the tree, since each node can have 2^k children, for k dimensions. At the same time, the maximum depth of the tree is independent of k and equal to the number of bits in the longest stored value, i.e. 8 when storing byte values. The PH-tree is expected to scale very well with large data sets, in some cases larger sets with $N > 10^6$ actually perform better than smaller data sets [Zäschke et al. 2014].

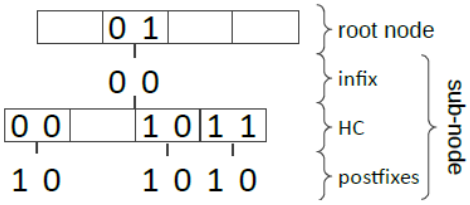

Figure 2. A representation of a 2D PH-tree with 3 4-bit inputs: (0001, 1000), (0011, 1000), (0011, 1010). Source: [Zäschke et al. 2014]

This algorithm is based on binary PATRICIA-tries combined with hypercubes. With hypercubes once the values of a k dimensional point have been interleaved into a bit-stream, they require only a constant time operation, i.e. an array look-up, to navigate to the sub-node or stored entry which allows efficient access to the data. The PH-tree is a multidimensional tree that uses hypercubes, prefix shares, and bit stream storage, where values can be stored in a way that uses exactly the number of bits they need. For example, storing a boolean requires only a single bit. The PH-tree understands only bit-strings, which it sorts as if they were integer values.

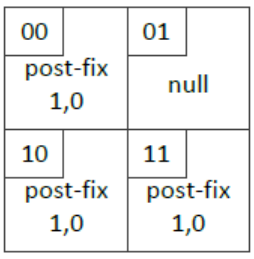

Figure 3. Array Hypercube (AHC), where the position numbers are addresses in the hypercube Source: [Zäschke et al. 2014]

The PH-tree has several interesting properties, such as the tree structure being independent of the insertion order, so insertion, update, or deletion will not affect more than two nodes.

3.1.3. Quad-tree

Quad-tree[Finkel and Bentley 1974] is a two-dimensional data structure. It's an hierarchical data structure in which the basic property it's recursive decomposition of space. This kind of structure store two-dimensional or even n-dimensional, in this case different names are used to call this data structure for example octree.

The Quad-tree it's also unbalanced data structure like PH-tree, that it's used to store a set of geospacial points and index then. An example of representation from this data structure is shown on figure 4.

The indexing process are created by the successive subdivision of all set of points known as quadrant into four sub quadrant. The quadrants are usually named according to their direction relative to the center of the parent node for example: *NW*, *NE*, *SW* and *SE*.

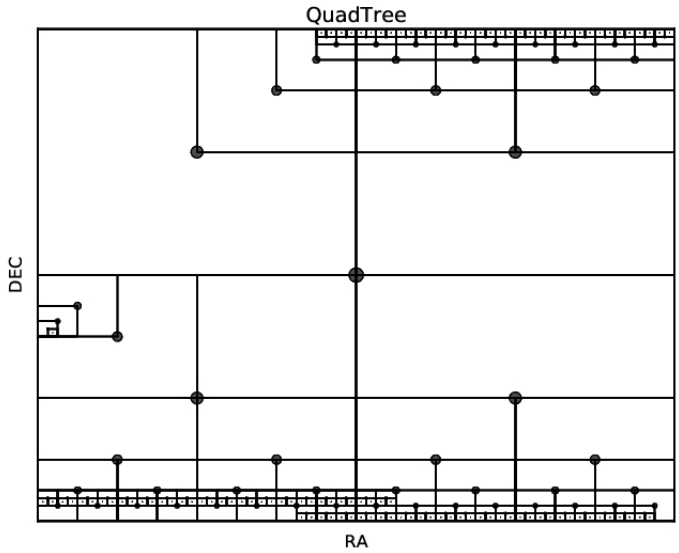

Figure 4. Quad-tree and it's recursive partitioning and it's unbalanced structure Source: [Porto et al. 2018b]

The successive subdivision occur until a certain level maximum height of the tree is reached. In this process besides maximum height of tree, the structure attempts to keep the same number of elements in all leaf nodes, even if they have different heights.

3.2. Metric Spaces and Distance Functions

To figure out how different the elements involved in the query are for data structures to execute queries by similarity and to return elements that most resemble a query element, a distance function is defined to calculate the distance between objects. [Santos 2012]

Distance functions create a metric space when they meet the properties of symmetry, non-negativity and triangular inequality, associated with a data domain.

A metric space is a pair (M, d), where M is a non-empty set and let $d : M \times M \rightarrow R$ be a distance or metric function that expresses the distance between the elements of M and satisfies the following properties:

(1) $d(x, x) = 0$

(2) $d(x, y) > 0$ if $x \neq y$

(3) $d(x, y) = d(y, x)$

(4) $d(x, y) \leq d(x, y) + d(y, z)$, $\forall x, y, z \in M$

If these properties are satisfied, d is a metric in M.

The Minkowski distance functions (Lp), are employed in multidimensional spaces. In this way, the geometrical distances of the Lp family are used to compare fixed-dimension data. Among the members of the Lp family is the euclidean distance.

In Euclidean geometry the smallest distance between two points is a straight line and the set of points at the same distance from a center forms a circle, according to euclidean distance.

3.3. Range Query on Spatial Indexing

A range query is a similarity query algorithm that can be used in all metric and spatial trees. Similarity queries search for elements in a dataset that are similar to a given query element. For similarity query operators to be applicable to a particular data domain, a distance function must be defined in that domain.

There are two basic types of query by similarity:

- Range Query;

- Nearest K-neighbors;

A range query works so that given a center point and a distance of dissimilarity, all neighbors found within the radius are returned. When the degree of query dissimilarity is 0, the range query is called a point query.

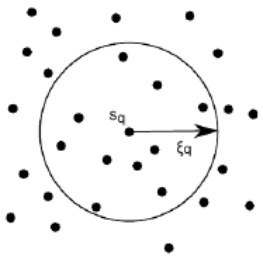

Figure 5. Range Query Source: [Santos 2012]

4. Implementation

Our experiments evaluated the following implementations of the studied data structures: a Slim-tree written in C ++; a Quad-tree written in Python and developed at DEXL Lab, and a PH-tree version 2.0.2, written in Java [3]. The Slim-tree implementation was based on the Arboretum project of the ICMC / USP Data Bases and Images Group (GBDI), version 1.0 [4].

[3]available at git-hub: https://github.com/tzaeschke/phtree

[4]available at: https://bitbucket.org/gbdi/arboretum/src/master/

4.1. Execution Environments and Datasets

In the first experiment, we run a range query that searches for 100 objects in the query space and considering 100 distances, both randomly generated. Thus, each object searches for neighbors over the 100 distances. The queries were applied to a dataset of 1.035.204 objects of the SDSS SkyServer DR12 project. All spatial objects were represented on datasets by attributes such as ID, spatial coordinates and other characteristics such as color. The experiment was run on a local machine using the Ubuntu 18.04 operating system, with 7.7GB of usable memory, Intel® Core™ 2 Duo processor. The execution of the programs was done sequentially. The algorithms for each test were run 30 times.

For the second experiment, the objective was to analyze the performance of the structures over data sets with different densities. For this purpose, a point was indexed in the center of the space where the data set was created with defined limits in relation to the position of the point and the distance. Ten data sets were created within these limits: $\rho 1 = 10^3$ elements, $\rho 2 = 10.10^3$ elements, $\rho 3 = 20.10^3$ elements up to $\rho 10 = 90.10^3$.

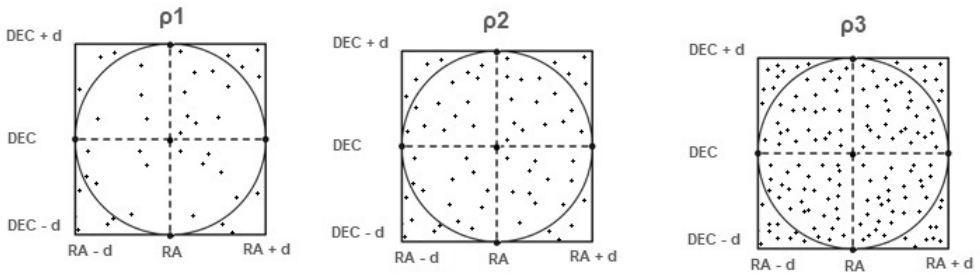

Figure 6. Representation of the same space with different data densities

The programs were executed in a local machine using the Ubuntu 18.04 operating system, with 15.5 GB of usable memory, Intel® Core™ i7-8700 CPU. The execution of the programs was done sequentially. The algorithms for each test were run 10 times.

5. Experiment Results

This section will be divided into two parts: Using a Dataset of the Order of 10^9 and Comparing Performance with Different Densities.

5.1. Using a Data set of the Order of 10^9

During the evaluation of these data structures, two experiments were carried on. The first one measures the behavior of the three data structures using the criteria: creation elapsed-time; search time and memory cost. The second one measures the accuracy of the results using the precision and recall metrics.

As depicted in figure 7, the PH-tree is significantly faster than the other competitor structures when building their in-memory structures. This is due to the use of prefix shares and bit stream storage. This leads to a faster process compared to other data structures that take objects in memory instead of binary key representation as index. The variance values were 0.00321889 seconds for PH-tree, 2.18315 seconds for Slim-tree and 0.0459924 seconds for Quad-tree.

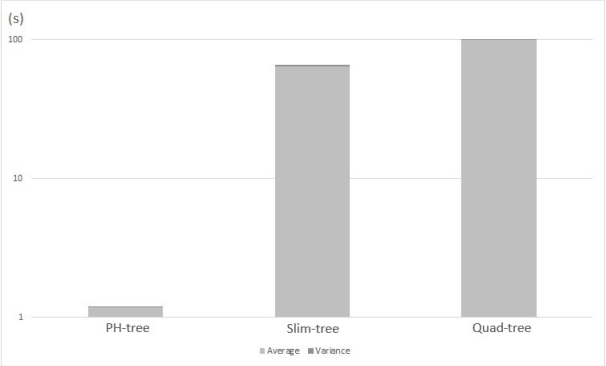

Figure 7. Average tree creation time with 1.035.204 objects - logarithmic scale

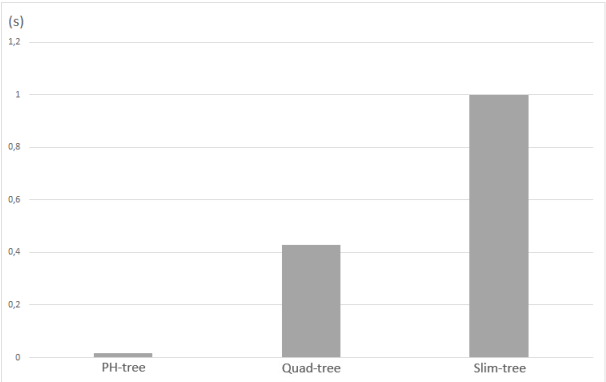

Figure 8. Average search time for neighbors over 100 distances. The values of PH-tree and Quad-tree are relative to the value of Slim-tree = 1.

In order to assess the accuracy of the neighboring queries among the three structures, we ran a controlled experiment having a data set with 98 objects. It was observed that all structures showed efficiency in neighborhood querying and corresponded to the expected positive values. However, Slim-tree presented better performance, with F-measure of 1.0, while Quad-tree 0.98846 and PH-tree 0.97005.

In Figure 8, we depict the average elapsed-time of neighboring queries for randomly selected 100 points and considering 100 randomly chosen distances. In average, PH-tree is a clear winner, 40% faster than Quad-tree, which in its turn is 120% faster than the Slim-tree. This, however requires further investigation. Thus, in Figure 9 we plot the average elapsed-time by query center point. PH-tree results are shown in solid lines and Slim-tree results are depicted in light dots.

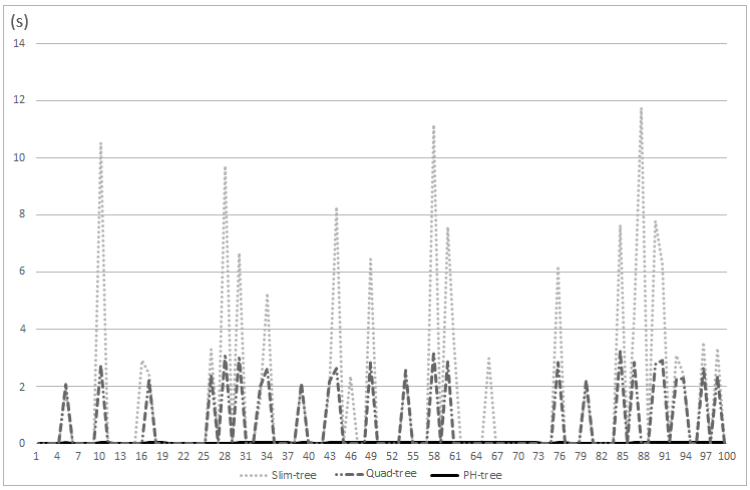

Figure 9. Average search time for neighbors with 100 distances for each object.

Note in Figure 9 that the average query elapsed-time time for the Slim-tree reaches very high peaks, up to 12 seconds. The Quad-tree also follows the pattern, although with up to 6x faster than the Slim-tree, This is probably due to the fact that the Slim-tree may require extra work as more tree entry regions intersect with each other. For very dense regions, this may happen often, leading to increasing number of paths to follow down the tree. In all cases, PH-tree keeps the query elapsed-time bellow 1 second.

The memory usage evaluation considered the same execution environment as the second experiment described in section 4.1. The results are shown in Figure 10. Despite the fact that the PH-tree was shown as the fastest during creation, memory footprint was considerably smaller for the Slim-tree, in almost two orders of magnitude. PH-tree may end up, in a worst case scenario, using one entry for each input key, if prefixes are not shared. This could be the case of *ra* and *dec* that are float numbers in a dense spatial region. The resident set size (RSS) is the amount of physical memory space (RAM) maintained by a process.

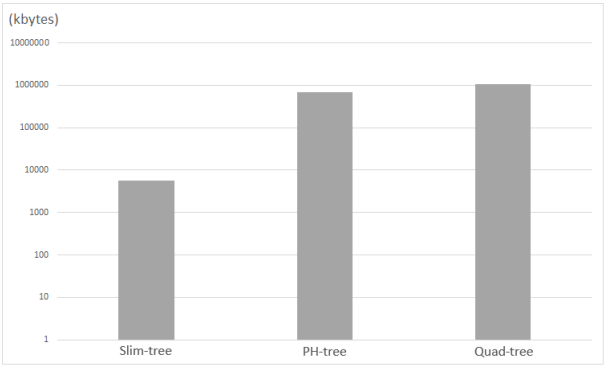

Figure 10. Maximum resident set size - logarithmic scale

5.2. Comparing Performance with Different Densities

As depicted in Figure 11, the Slim-tree showed the shortest creation time when considering a limited amount of input objects. As the density of objects in a given region increases,

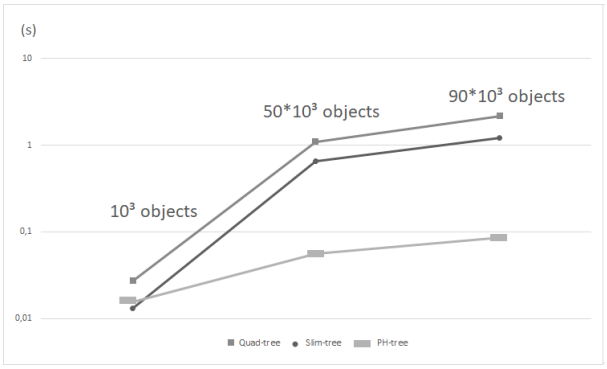

Figure 11. Average tree creation time with different densities - logarithmic scale

PH-tree becomes more efficient. The reason may be that as the number of objects increases, Slim-tree needs to split more often its internal nodes. The PH-tree structure does not require node splitting showing negligible impact with increasing number of objects.

The standard deviation was low, being approximately 0.0013 seconds for Slim-tree with 10^3 objects, 0.65 seconds for Slim-tree with $50 * 10^3$ objects and 1.21 seconds for Slim-tree with $90*10^3$ objects. For PH-tree the standard deviation was 0.0015 seconds with 10^3 objects, 0.003 seconds with $50 * 10^3$ objects and 0.003 seconds with $90 * 10^3$ objects. The standard deviation for Quad-tree was 0,012 seconds with 10^3 objects, 0.016 seconds with $50*10^3$ objects and 0,032 seconds with $90*10^3$ objects. Thus, as data scales up, PH-tree becomes a clear winner. This corroborates the results depicted in Figure 7.

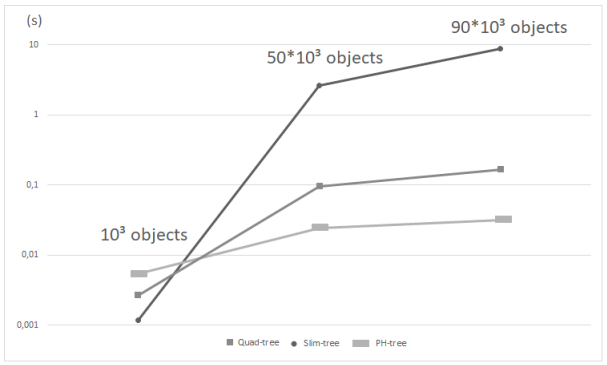

Figure 12. Average search time with different densities - logarithmic scale

For querying neighbors under different densities, Slim-tree and Quad-tree were more efficient for a small amount of objects, but once again PH-tree proved to be more efficient as the amount of data increased. This may be associated to the same issue raised before, associated with increasing nodes intersections. As for Quad-trees, dense areas extend the tree height leading to multiple recursive node navigation.

The results for standard deviation was approximately 0,0001 seconds for Slim-tree with 10^3 objects, 0.05 seconds for Slim-tree with $50 * 10^3$ objects and 0.23 seconds for Slim-tree with $90 * 10^3$ objects. For PH-tree the standard deviation was 0.0008 seconds with 10^3 objects, 0.0008 seconds with $50 * 10^3$ objects and 0.0015 seconds with $90 * 10^3$ objects. The standard deviation for Quadtree was 0.0003 seconds with 10^3 objects, 0.0011 seconds with $50 * 10^3$ objects and 0.0011 seconds with $90 * 10^3$ objects.

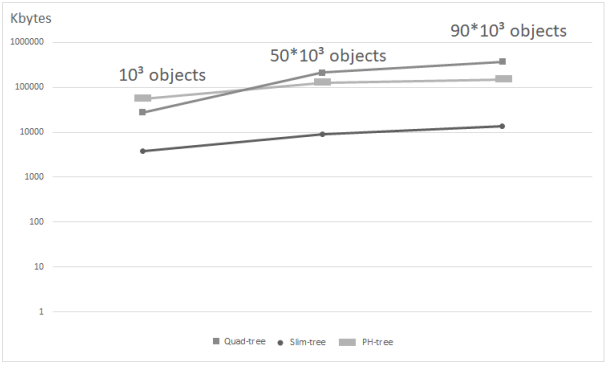

Figure 13. Average maximum resident set size with different densities - logarithmic scale

The results referring to memory footprint with different data densities reproduced that of Figure 10. In Figure 13, Slim-tree maintains a steady allocation approximately one order of magnitude smaller than the two others.

6. Related Works

The extension of Database Management Systems to support spatial applications is not new. As more spatial data become prevalent due to sensor capturing in various sources and the release of spatially indexed scientific data sets, efficient spatial data processing become paramount. SpatialHadoop [Eldawy and Mokbel 2015] is probably the first Big Data framework to incorporate spatial data types, indexes and support for SQL/MM. The Hadoop framework and all the others built on top of its infrastructure suffered from poor performance, mostly due to the simple Map/Reduce interface and the disk based storage of intermediate results. More recently, there has been systems developed on top of Apache Spark. GeoSpark is a complete implementation of spatial data types, spatial data partitioning algorithms, spatial indexing and spatial SQL [Yu et al. 2019]. The system supports range queries over data that can be spatially indexed using either R-tree [Guttman 1984] or Quad-tree. SpatialSpark [You et al. 2015] is another system built on top of Apache Spark. It extends the RDD in-memory datatype to support R-tree data partitioning and R-tree indexing of partitions. In fact, to the best of our knowledge, none of the state-of-the art extensions of Big Data frameworks, such as Apache Spark, has been extended with the PH-tree or Slim-tree indexing structures in support of efficiently solving spatial range queries. Despite that, the approach of building trees for each RDD partition, as in GeoSpark, follows exactly the same partitioning approach taken in this work.

7. Conclusion

In this paper, the study of data structures that perform data indexing from a database and perform queries based on a neighborhood relation, showed that these structures are very different in relation to the use of memory. Slim-tree allocates less memory than PH-tree and Quad-tree. For frameworks such as Apache Spark, a reduced use of memory is relevant. On the other hand, PH-tree presented a significantly better performance, as the volume of data increased, in terms of the construction time of its structure, as well as in the search time of objects in different densities under a defined radius compared to other structures. Its binary structure and queries spanning bit stream have shown a much

shorter insertion time and data access than the other structures, showing itself to be the best method for conducting neighborhood queries in SDSS dataflows on Apache Spark.

References

Eldawy, A. and Mokbel, M. F. (2015). The ecosystem of spatialhadoop. *SIGSPATIAL Special*, 6(3):3–10.

Finkel, R. A. and Bentley, J. L. (1974). Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9.

Guttman, A. (1984). R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM International Conference on Management of Data - SIGMOD*, pages 47–57. ACM.

Khatibi, A., Porto, F., Rittmeyer, J. G., Ogasawara, E., Valduriez, P., and Shasha, D. (2017). Pre-processing and indexing techniques for constellation queries in big data. In *Proceedings of the International Conference on Big Data Analytics and Knowledge Discovery*, pages 164–172. Springer International Publishing.

Ozsu, T. and Valduriez, P. (2011). *Principles of Distributed Database Systems*. Springer Verlag.

Porto, F., Khatibi, A., Rittmeyer, J. N., Ogasawara, E. S., Valduriez, P., and Shasha, D. E. (2018a). Constellation queries over big data. In *XXXIII Simpósio Brasileiro de Banco de Dados, SBBD 2018, Rio de Janeiro, RJ, Brazil, August 25-26, 2018.*, pages 85–96.

Porto, F., Rittmeyer, J., Ogasawara, E., Krone-Martins, A., Valduriez, P., and Shasha, D. (2018b). Point pattern search in big data. In *Proceedings of the 30th International Conference on Scientific and Statistical Database Management*, page 21. ACM.

Ramakrishnan, R., G. J. (2002). *Database Management Systems*. McGraw-Hill.

Samet, H. (1984). The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260.

Santos, L. F. D. (2012). *Explorando variedade em consultas por similaridade*. PhD thesis, Universidade de São Paulo.

Traina, C., Traina, A., Seeger, B., and Faloutsos, C. (2000). Slim-trees: High performance metric trees minimizing overlap between nodes. In *International Conference on Extending Database Technology*, pages 51–65. Springer.

You, S., Zhang, J., and Gruenwald, L. (2015). Large-scale spatial join query processing in cloud. In *31st IEEE International Conference on Data Engineering Workshops*. IEEE.

Yu, J., Zhang, Z., and Sarwat, M. (2019). Spatial data management in apache spark: the geospark perspective and beyond. *GeoInformatica*, 23(1):37–68.

Zäschke, T., Zimmerli, C., and Norrie, M. C. (2014). The ph-tree: a space-efficient storage structure and multi-dimensional index. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 397–408. ACM.