Use-based Optimization of Spatial Access Methods

Nikolaos Athanasiou
Dept. of Informatics, Aristotle University
Thessaloniki, Greece
n.g.athanasiou@gmail.com

Michael Vassilakopoulos
Dept. of Electrical and Computer Eng.
University of Thessaly, Volos, Greece
mvasilako@uth.gr

Antonio Corral
Dept. of Informatics, University of Almeria
Almeria, Spain
acorral@ual.es

Yannis Manolopoulos
Dept. of Informatics, Aristotle University
Thessaloniki, Greece
manolopo@csd.auth.gr

ABSTRACT

Spatial access methods have been extensively studied in the literature, during last decades. Access methods were designed for efficient processing of demanding queries and extensive comparisons between such methods have been presented. However, choosing the best values for the parameters that affect the performance of a spatial access method when such a method is expected to be utilized within a specific workload / context of operations has not been studied, so far. In this paper, we present the design and implementation of a framework to evaluate and optimize a spatial index. The (very popular) family of R-trees is chosen as the index of focus, though the same process can be applied for other (spatial, or not) indexes or combinations of them. We elaborate on the antagonizing aspects in the design of an R-tree, present the design and implementation of a benchmarking framework and develop a performance model for this index that incorporates benchmarking results. Next, we develop an optimization framework that uses this model to provide an optimized set up (node occupancies and node splitting method) for a specific use context (dataset type, tree size, number of queries and type of queries). We also present experimental results of an indicative use of the developed benchmarking framework and optimizer for a limited range of use contexts.

CCS CONCEPTS

• **Information systems → Data access methods**; • **Theory of computation → Data structures and algorithms for data management**;

KEYWORDS

Spatial Access Methods, Performance, Optimization

MEDES '17, November 7–10, 2017, Bangkok, Thailand

ACM ISBN 978-1-4503-4895-9/17/11…$15.00
https://doi.org/10.1145/3167020.3167030

1 INTRODUCTION

Due to the demanding need for efficient spatial database applications, significant research effort has been devoted to the development of efficient spatial index structures. However, as shown in several previous performance comparative studies [9, 19], there is no unique spatial index structure that works efficiently, in all cases. On the other hand, it is well-known that hierarchical index structures are useful because of their ability to focus on the interesting subsets of data. This focusing results in an efficient representation and improved execution times on spatial query processing. Important advantages of these index structures are their conceptual clarity and their capability for processing a diverse range of spatial queries.

Current and future spatial database systems should include versatile and configurable Spatial Access Methods (SAMs) to answer spatial queries. Such access methods will allow to dynamically tune different parameters during building of the method and/or before the execution of specific spatial queries. Tuning SAMs becomes important if we consider the performance of an access method during its lifetime.

A workload is an input set of parameters for the analysis of the metrics that characterize the performance of an index structure. In our case, the workload is a dataset size and distribution for which the access method is initially built and a set of queries to be executed before rebuilding the access method. A specific workload allows an optimizer to tune an index by selecting the best possible set-up of the index for this workload. That is, such an optimizer can calculate performance metrics for each workload, using different parameters that affect the structural characteristics of the access method (for example, *min* and *max* branching factors for the nodes of the index and splitting algorithms of overflown nodes) and reveal the ones that lead to the best performance. To measure the performance of the access method using several choices of parameters values for specific workloads the design and implementation of a benchmarking framework is necessary. The use of such an optimizer is very important for applications and the industry to obtain good performance for specific workloads. For example, a service like Foursquare could improve its performance by optimizing the access methods it uses, depending on its expected use for specific areas, points-of-interest and user queries.

In this paper, we address the following problem: *to implement a use-based optimizer capable of choosing the best parameters (as branching factors and node split algorithms) for creating an R-tree*

index for efficient spatial query processing. Thus, we present the development and use of such an optimizer that is based on cost estimation (total execution time, or latency) and chooses the best split algorithm for overflown nodes and *min* and *max* node branching factors to build an R-tree and subsequently perform a series of specific spatial query operations. The proposed use-based optimizer could be embedded as a special module in the general query optimizer that a spatial DBMS incorporates.

The contribution of this paper is summarized as follows. By focusing on the (very popular) family of R-trees, though the same process can be applied for other (spatial, or not) indexes or combinations of them,

- we present the design and implementation of a benchmarking framework for R-trees,
- we develop a performance model for this index that incorporates benchmarking results, and
- we develop a swarm optimization framework that uses this model to provide an optimized R-tree set up (node occupancies and node splitting method) for a specific use context (dataset type, tree size, number of queries and type of queries).

In the rest of this paper, we present related work and motivation for this research (in Section 2), the R-tree required background (in Section 3), a description of the benchmarking framework we developed (in Section 4), the foundation and the functioning of the optimizer we developed (in Section 5), the results of indicative experimentation performed (in Section 6) and, finally, the conclusions arising from this work and our future research plans (in Section 7).

2 RELATED WORK AND MOTIVATION

Database optimization refers to a variety of strategies for reducing database system response time, and it allows a better configuration and faster searches results. This optimization involves maximizing the speed and efficiency based on the data to retrieve. Database designers, administrators and analysts work together to optimize system performance through diverse methods and techniques. One of these techniques is the cost estimation that applies consistent and significant performance measures of the costs for specific indexes and queries. Different metrics can be used for this purpose, but the most relevant and common metric is the number of page accesses. Since disk I/Os represent time-consuming operations, the objective is to minimize the number of block accesses without the sacrifice of functionality. Other performance metrics are CPU time and latency; they are the most representative from the user point of view.

In the literature, several SAMs have been compared. For instance, in [19], apart from an initial insertion cost of several SAMs, the benchmark focused on query performance. In [9], a number of comparative studies of multidimensional indexes and attempts to unify the results into a partial ordering of quality are presented. Another case is published in [13], where the main objectives are to provide a benchmarking environment for SAMs and related query evaluation techniques, and to allow comparative studies of SAMs in different cases but under a common framework. In [17], an analysis framework for tree-structured balanced access methods that can be used to evaluate the *page access* performance of user-defined query workloads is also presented.

As we have exposed, traditional benchmarks consider comparisons of different access methods with respect to the query processing costs, or even only the cost of specialized query processing costs [8, 12, 15]. On the other hand, [25] introduces a new benchmark that captures metrics about online indexing techniques. The main distinction of [25] is that it includes the index creation costs as part of the processing performance, and is thus suited for evaluating on-line techniques that create indexes on-the-fly. The consideration of the index creation cost as part of the performance can also be taken into account in optimizations of spatial indexes.

The most crucial part of the R-tree construction is the node splitting procedure because of overfkow during insertion. The performance of R-tree is influenced by mainly two important factors: coverage and overlap. There have been several R-tree node splitting algorithms developed since R-tree was introduced. Among them, the most popular ones are: Guttman's Quadratic-Cost algorithm [14], the R*-tree splitting algorithm [4], the Optimal node splitting algorithm [10], the Linear node splitting algorithm [1] and Basic node splitting with bipartition [21]. And as we know, a good node splitting algorithm should confirm the minimum overlapping area among the Minimum Bounding Rectangles.

Recently, new ideas on how to build access methods that have tunable behavior and design trade-offs have been proposed [2]. Moreover, a close look at existing proposals on access methods reveals that each method is confronted with the same fundamental challenges and design decisions again and again. In particular, there are three quantities and design parameters that researchers always try to minimize: (i) the read overhead (R), (ii) the update overhead (U), and (iii) the memory (or storage) overhead (M), henceforth called the RUM overheads or RUM conjecture [3]. Deciding which overhead(s) to optimize for and to what extent remains a prominent part of the process of designing a new access method, especially as hardware and workloads change over time.

Although, these research efforts head in the direction of creating more efficient access methods, a complete optimizer that leads to specific choices for the key parameters of a tree based index, considering its use during its lifetime has not been created yet. In this paper, we present such an optimizer, as a first approach to address an existing need of the industry for workload optimization of SAMs.

3 BACKGROUND ON R-TREES

R-trees [14] are hierarchical, height balanced data structures, derived from B-trees [6] and designed to be used in secondary storage. R-trees are considered as excellent choices for indexing various kinds of spatial data (points, rectangles, line-segments, polygons, etc.) and have been adopted in known commercial systems (e.g. Informix [5], Oracle Spatial [11, 18], MySQL [26], PostGIS [7, 23], etc.). They are used for the dynamic organization of a set of spatial objects represented by their Minimum Bounding Rectangles (MBRs). The MBR represents the smallest axes-aligned rectangle in which the spatial objects are contained. A 2d MBR is determined by two 2d points that belong to its faces, one that has the minimum and one that has the maximum coordinates (these are the endpoints of one of the diagonals of the MBR). Using the MBR instead of the exact geometrical representation of the object, its representational

complexity is reduced to two points where the most important features of the spatial object (position and extension) are maintained. The MBR is an approximation widely employed, and the R-trees belong to the category of *data-driven access methods* [24], since their structure adapts to the MBRs distribution in space (the partitioning adapts to the object distribution in the embedding space).

The rules obeyed by the R-tree are as follows:

(1) All leaves reside on the same level.
(2) Each leaf node contains entries, E, of the form *(MBR, Oid)*, such that *MBR* is the minimum bounding rectangle that encloses the object determined by the identifier *Oid*.
(3) Internal nodes contain entries, E, of the form *(MBR, Addr)*, where *Addr* is the address of the child node and *MBR* is the minimum bounding rectangle that encloses MBRs of all entries in that child node.
(4) Nodes (except possibly for the root) of an R-tree of class *(m, M)* contain between m and M entries, where $m \leq \lceil M/2 \rceil$ (M and m are called maximum and minimum branching factor, or fan-out).
(5) The root contains at least two entries, if it is not a leaf.

Figure 1 presents a set of 2-d points, along with their grouping into MBRs and the corresponding R-tree.

As long as the number of objects in each R-tree leaf node is between m and M, no action needs to be taken on the R-tree structure other than adjusting the bounding boxes when inserting or deleting an object. If the number of objects in a leaf node decreases below m, then the node is said to underflow. In this case, the objects in the underflowing nodes must be reinserted, and bounding boxes in non-leaf nodes must be adjusted. If these non-leaf nodes also underflow, then the objects in their leaf nodes must also be reinserted. If the number of objects in a leaf node increases above M, then the node is said to overflow. In this case, it must be split and the contained M+1 objects must be distributed in two resulting nodes. Splits are propagated up to the tree root.

The efficiency of the R-tree for search operations depends on its ability to distinguish between occupied and unoccupied space and to prevent a node from being examined needlessly due to an empty overlapping subregion with other nodes. The extent of success of this distinguishing ability is a direct result of how well we can satisfy the goals of minimizing coverage and overlap. Guttman [14] proposed three alternative algorithms of different complexity to handle splits:

- Linear. Choose two objects as seeds for the two nodes, where these objects are as furthest as possible. Then, consider each remaining object in a random order and assign it to the node requiring the smaller enlargement of its respective MBR.
- Quadratic. Choose two objects as seeds for the two nodes, where these objects if put together create as much dead space as possible (dead space is the space that remains from the MBR if the areas of the two objects are ignored). Then, until there are no remaining objects, choose for insertion the object for which the difference of dead space if assigned to each of the two nodes is maximized, and insert it in the node that requires smaller enlargement of its respective MBR.
- Exponential. All possible groupings are exhaustively tested and the best is chosen with respect to the minimization of the MBR enlargement

Many variations of R-trees have appeared in the literature (extended surveys can be found in [9, 22]). One of the most popular and efficient variations is the R*-tree [4]. The latter structure is a variant of R-trees that does not obey the limitation for the number of pairs per node and follows a sophisticated node split technique. More specifically, the technique of "*forced reinsertion*" is applied, according to which, when a node overflows, p entries are extracted and reinserted in the tree (p being a parameter, with 30% its suggested optimal value). Another novel feature of the R*-tree is that it considers additional criteria except the minimization of the sum of the areas of the produced MBRs. These criteria are the minimization of the overlapping between MBRs at the same level, as well as the minimization of the perimeter of the produced MBRs.

In [1], a linear algorithm to distribute the objects of an overflowing node in two sets has been proposed. The primary criterion of this algorithm is to distribute the objects between the two nodes as evenly as possible, whereas the second criterion is the minimization of the overlapping between them. Finally, the third criterion is the minimization of the total coverage. Experiments using this algorithm have shown that it results in R-trees with better characteristics and better performance for window queries in comparison with the quadratic algorithm of the original R-tree.

A sort-based method for bulk-loading R-trees, called *Sort-Tile-Recursive* (STR) method, is presented in [20], and it can be applied to d-dimensional datasets. Let's consider, without loss of generality, that the underlying space is two-dimensional, although the extension of the method to higher dimensions is straightforward. Let's assume a total of r rectangles and a node capacity of n rectangles per leaf node, the set of leaf nodes of the tree is formed by constructing a tiling of the underlying space consisting of $\sqrt{r/n}$ vertical slabs, where each slab contains $\sqrt{r/n}$ tiles, or leaf nodes (each leaf node is filled to capacity). This way the underlying space is being tiled with rectangular tiles, where, like a grid, vertically adjacent tiles (i.e., with a common horizontal edge) are aligned, but unlike a grid, horizontally adjacent tiles (i.e., with a common vertical edge) are not aligned, i.e. their horizontal edges do not form a straight line. Therefore, the underlying space is tiled with approximately $\sqrt{r/n} \times \sqrt{r/n}$ tiles and partitioned in approximately r/n leaf nodes. The tiling process is applied recursively to the dataset corresponding to these r/n leafs to form the sets of nodes of the previous levels until just one node is obtained.

4 BENCHMARKING FRAMEWORK

A C++ benchmarking framework was developed for the needs of this paper. Its goal is to provide a generic, cross platform (standard compliant), non-dependent to third party libraries solution to the benchmarking problem. The main features of this design are:

- parameterizable time unit per benchmark,
- automated running of multiple experiments on the same benchmark (hence, results can be grouped and compared, as wished),
- parameterizable number of repetitions (sampling size) of each experiment,

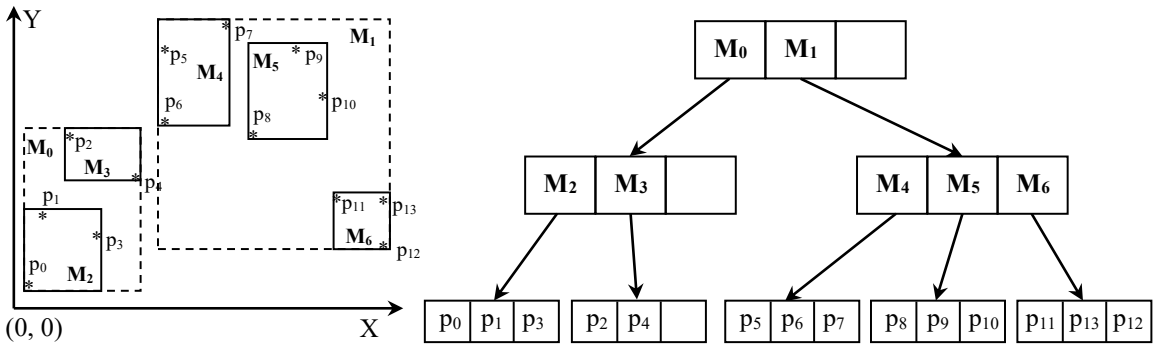

Figure 1: A set of 2-d points and their grouping into MBRs (left) and the corresponding R-tree (right).

- parameterizable rate of increase of the basic variable of the experiments (the variable shown along the x axis, when results are presented as diagrams), and
- benchmark results printable and serializable in a "Python friendly" form.

As a complementary tool, a Python script was developed, to read the output of a serialization and make diagrams.

To minimize the effect of hardware and OS interference in the benchmarking framework, we took a number of precautions. A cache cleaning process runs before every measurement and effectively displaces previous experiments data from the cache by allocating and processing a big chunk of memory, set by default to 20MB to handle even L3 caches on typical home computers. Also, after each experiment a `rest()` command is executed so that the next experiment does not take advantage of the results of the previous one. To avoid context switching imposed by the OS, the benchmark execution waits for the user to set the thread affinity to a single core from the (Windows) Task Manage and then proceeds to the experiments.

A visualization package was also developed to complete the benchmarking framework. The benchmarks are serializable as Python dictionaries. This way the visualization scripts can translate them into plots using `matplotlib` and `pyplot`. Additionally, a minimal GUI was added to facilitate file operations, using the `tk` Python library.

Different experiment types for loading and querying were created. The query experiments, use the R-trees constructed by the loading experiments.

5 THE R-TREE OPTIMIZER

This section proposes a strategy for algorithmically optimizing the usage of an R-tree data structure. By "algorithmically optimizing" we refer to the process of selecting the optimized *min* and *max* limits of node occupancy and splitting algorithms for a given context.

Optimizing the usage of the R-tree data structure requires abstracting the elements that determine its performance. To this end, we identify the following factors:

- Size: The number of data elements in the R-tree.
- Query rate: Average queries performed for every alteration (node insertion / removal) of the R-tree, i.e. how many times, on average, we query the tree before modifying it.
- Query Construction rate: Average queries performed before reconstructing the R-tree.
- Dataset distribution: The characteristics of the space we operate on.

The following simple formula to express the latency (total execution time) of an R-tree emerges:

$$l = \sum^{N_r} \left(t_n + \sum^{N_\alpha} (Q_r \times t_q + t_a)\right)$$

where:

- l= the overall time of operation (usage) of the tree,
- N_r=the number of reconstructions,
- t_n=time to construct a tree of n entities,
- N_a=number of modification operations (updates / insertions / deletions) to the tree,
- Q_r=the query rate,
- t_q=average query time, and
- t_a=time needed for a modification operation to the tree.

To create a performance model of the data structure, we decided to consider a single life-cycle (i.e. from construction until reconstruction) and replace the query rate by a factor named "*number of queries*":

$$t = t_n + N_q \times t_q + t_a$$

where:

- t=time of operation (objective function),
- t_n=construction time for n elements,
- N_q=number of queries,
- t_q=query time, and
- t_a=time needed for a modification operation to the tree

With this simplified version, t is the objective function, i.e. the latency is the target property to minimize. We are now ready to build a program that generates an "R-tree life-cycle" (workload). The values of the right part of the above formula depend on the distribution (morphology) of the dataset and the number and type of queries performed. Therefore, our program is parameterizable by the following factors:

- Input dataset,

- R-tree size (number of data elements),
- Number of queries, and
- Query type

and the target properties to optimize are:

- *min* node occupancy,
- *max* node occupancy, and
- Node splitting algorithm (*linear, quadratic, R*, bulk*)

The results generator returns the time of operation; thus, it can be used as an objective function by the optimization packages as described in the following. In the indicative experiments that we performed, the context setup can take the following values:

- Dataset: *real2d, synthetic2d.*
- Number of data elements: integer value (less than the cardinality of the dataset for real datasets).
- Number of queries: unbounded positive integer value.
- Query type: *within, contains, covered by, covers, disjoint, intersects, kNN, overlaps.*

As stated before, the output of the optimizer consists of the best combination of *min* and *max* node occupancies and split algorithm (*linear, quadratic, R*, bulk*). The optimizer gives values to the elements we try to optimize and then decides which values gave the best behavior. In 1995, [16] built upon earlier investigations that suggested that members of a group can benefit not only from their individual memories, but also from the collective memory of the entire group. This is the underlying concept behind Particle Swarm Optimization (PSO). PSO is a computational method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality. It solves a problem by having a population of candidate solutions, dubbed particles, and moving these particles around in the search space according to simple mathematical formulas over the particle's position and velocity. Each particle's movement is influenced by its local best known position, but is also guided toward the best-known positions in the search-space, which are updated as better positions are found by other particles. This is expected to move the swarm toward the best solutions. The language of choice to build our optimizer is Python, due to the plethora of easy to use, freely available packages. The `pyswarm` package[1] is a gradient-free, evolutionary optimization package for python that supports constraints and implements PSO. `pyswarm` was used to build our optimizer.

5.1 Structure of the optimizer

To create the optimizer, an inter-language communication between C++ and Python had to be established. The components interconnect as shown in Figure 2.

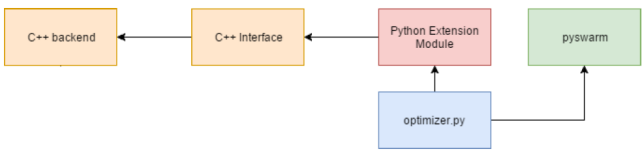

Figure 2: Layout of the optimizer.

In the above architecture, we see the following:

- a C++ backend that contains the implementation of the objective function, i.e. the generator of an R-tree "lifetime",
- a C++ interface that exposes the generator function,
- a Python extension module, i.e. a C++ library build to export functions to the Python programming language,
- the `pyswarm` package, and
- the Python-based optimizer module that drives the whole process.

In PSO, the objective function takes a candidate solution as an argument in the form of a vector of real numbers and produces a real number as output. This means the search space is assumed to be continuous while our parameters are not. For example, if the maximum node occupancy (*max*) is set to lie in the [4, 256] range, the minimum node occupancy (*min*) is a percentage of *max* in the range [0.1, 0.5]. We need to avoid re-evaluating the objective function over input that has been truncated to integer values, since in PSO it is not guaranteed that identical results will be produced for the same input. Therefore, we introduce a search space discretization process, implemented as a dictionary of precomputed values: for an example input of *max* =2.3 and *min* =1.15, we first check the dictionary for the pair of values (2,1) and use the stored value as a result, if found, or the compute the result for the pair of values (2,1) and store its result, if not. The optimizer generates the results of the cost function for the vectors of real numbers given as input (the optimization context) and the solution to this context is calculated using PSO on the results of the cost function. To speed up the optimization process, we utilize the multiple cores a computer possesses. Our processing is inherently parallel, since there is independence of the computation of the generating function for different discrete input vectors. This approach provides an almost linear speedup, without loss of quality (as the results of the numerous experiments we performed demonstrate). A sample output of the optimization process can be seen in Figure 3. We used a synthetic dataset to construct a tree of 50000 elements and performed 10000 "within" queries, using the various node splitting algorithms. The clear winner is a bulk loaded R-tree with (11, 40) of (*min, max*) node occupancy.

```
*** R-tree optimization with pyswarm (79.0 minutes) ***

=======================================================
dataset=syth2d, #Elems=50000, #Qs=10000, qType=within
=======================================================

optimum latency = 127 | nodes = ( 3,  6) | split = rstar
optimum latency =  78 | nodes = ( 2,  7) | split = qdrt
optimum latency =  58 | nodes = ( 3, 23) | split = lin
optimum latency =  14 | nodes = (13, 40) | split = bulk
```

Figure 3: Sample output of the optimizer.

6 EXPERIMENTATION

In this section we present the results of our experimental evaluation. This evaluation is intended to demonstrate the functioning of the developed optimizer and does not aim to exhaustively study the setup up of an R-tree for a broad context range. All

[1] `http://pythonhosted.org/pyswarm/`

experiments are conducted on a computer with the following characteristics: Intel(R) Core(TM) i5-4570 CPU @ 3.20GHz (4 cores), L1 Data Cache: 32 KB, Line Size 64 L1 Instruction Cache: 32 KB, Line Size 64, L2 Unified Cache: 256 KB, Line Size 64, L3 Unified Cache: 6 MB, Line Size 64, Physical memory (RAM): 24 GB. Both synthetic and real data were used to "feed" the experiments. The generation of random data was performed using C++11 <random> facilities to produce 2D boxes, uniformly distributed in space. Rectangular real datasets were created by reading the world map at `http://www.gadm.org/version2`. The R-tree implementation used is the package available in Boost Geometry[2], a highly parameterizable spatial index package written in heavily templated modern C++. For each experiment, the response time (performance metric) in seconds was recorded. We set the thread affinity of the executable to just use CPU 0 to avoid context switching and minimize OS interference. Loading an R-tree was performed iteratively, using the *linear*, *quadratic* and *R*-tree* split algorithms, as well as, by *bulk loading*. Execution of the *Contains, Covered by, Covers, Disjoint, Intersects, kNN, Overlaps*, and *Within* queries was performed on *Real* and *Synthetic* datasets. The search space of the optimizer, the interval of values for both *min* and *max*, was set to [4, 128] while the *Tree Size* ranged from 25K to 100K in steps of 25K and the *Query Size* (number of queries) ranged from 25K to 100K in steps of 25K.

Due to space limitations, we present the results of two series of experiments (note that each combination of dataset, tree size, number of queries and query type constitutes a different workload). In the first one, the *Tree Size* (number of elements in the tree) was fixed to 50K, for *Real* and *Synthetic* datasets. The *Query Size* (number of queries) ranged from 25K to 100K in steps of 25K. *Within* and *Overlaps* queries for all four *Splitting Strategies* were tested. In Figure 4, the results of this experiment are presented graphically. Each group of bars corresponds to a different value of number of queries. In each group, the four bars (from left to right) express the range of *min* and *max* occupancies (the scale of which is depicted on the left) of trees created by the *bulk*, *linear*, *quadratic* and *R*-tree* split algorithms, while, the diamond, triangle, quadrangle and circle express the *Latency*, or time of operation, (the scale of which is depicted on the right) of the trees created by the *bulk*, *linear*, *quadratic* and *R*-tree* split algorithms. Studying the results of this experiment, we note the following. For the *Synthetic* dataset, the *bulk* split algorithm has, by far, the best performance for both queries. For the *Real* dataset, the *bulk*, or *linear* split algorithm, depending on *Query Size*, has the best performance for both queries, although for the *Overlaps* query, the *quadratic* split algorithm is quite close.

In the second set of experiments, the *Query Size* was fixed to 50K and the *Tree Size* ranged from 25K to 100K in steps of 25K, for *Real* and *Synthetic* datasets. Again, *Within* and *Overlaps* queries for all four *Splitting Strategies* were tested. In Figure 5, as a second example, the results of this experiment are presented, in an analogous manner to Figure 4. Studying the results of this experiment, we note the following. Even for varying the *Tree Size*, for the *Synthetic* dataset, the *bulk* split algorithm still has, by far, the best performance for both queries. For the *Real* dataset, the *bulk*, or *linear* split algorithm, depending on *Tree Size*, has the best performance for the *Within*, while the *bulk* split algorithm is always the best for the *Overlaps* query. Again, for the *Overlaps* query, the *quadratic* split algorithm is quite close to the *bulk* and *linear* ones.

In both these experiments and in the rest of the experiments performed, the bad performance of the *R*-tree* split algorithm is attributed to the fact that the improved tree created by forced reinsertion is not paid off for the numbers of queries studied. Studying the complete set of experiments, we note that in the majority of cases, the absolute values and the range between *min* and *max* are larger for the *Real* dataset for all the queries, except for *Disjoint* and *kNN*. Moreover, we conclude that *Real* datasets are related to higher latency for the *Overlaps* query, while *Synthetic* datasets are related to higher latency for the *Disjoint* query. If the nature of the dataset is unknown, the time performance values of these queries can help us determine the type of dataset and accordingly adjust the optimizer search space for the interval of values for *min* and *max* (since, *Real* datasets require studying a wider range of values). Studying the complete sets of the experiments, it also is clear that the best R-tree set-up (the combination of *min* and *max* node occupancies and split algorithm) is heavily related on the context (dataset type, tree size, number of queries and type of queries) within which the tree will be used.

7 CONCLUSIONS AND FUTURE WORK

In the paper, a use-based optimizer of R-trees (popular access methods) was presented. This optimizer can lead to a set-up of an R-tree that maximizes performance, considering the tentative use of this index in an application context. Such an optimizer would be very useful for developers, since, in most cases, they are not interested in the performance of an access method for specific queries only, but in the overall performance during its lifetime in an application. The first conclusion that became apparent is that there is "no single solution" for the settings of a spatial index (at least for the popular family of R-trees). Specific settings of the R-tree are better suited for its different uses. While performance is clearly superior for specific methods, e.g. load time with the STR method, it is the overall lifetime of the structure that reveals the best technique to use (e.g. static trees would pay for rebuilding and contexts where we have frequent modifications to the R-tree would require a dynamic version).

Suggestions for future work include:

- to extend the optimizer, by using more optimization methods, or to reduce its runtime by deducing the query times by sampling,
- to create and use more complex, realistic and parametric lifetime models that better reflect real application contexts,
- to consider more datasets and to extend the benchmarking framework to incorporate user-specified datasets,
- to consider the implementation and use of more node splitting algorithms,
- to evaluate more spatial indexing methods,
- since, benchmarking process takes a time proportional to the size of the input dataset, to consider how to further parallelize experiments without loss of accuracy, and

[2] `http://www.boost.org/doc/libs/1_60_0/libs/geometry/doc/html/index.html`

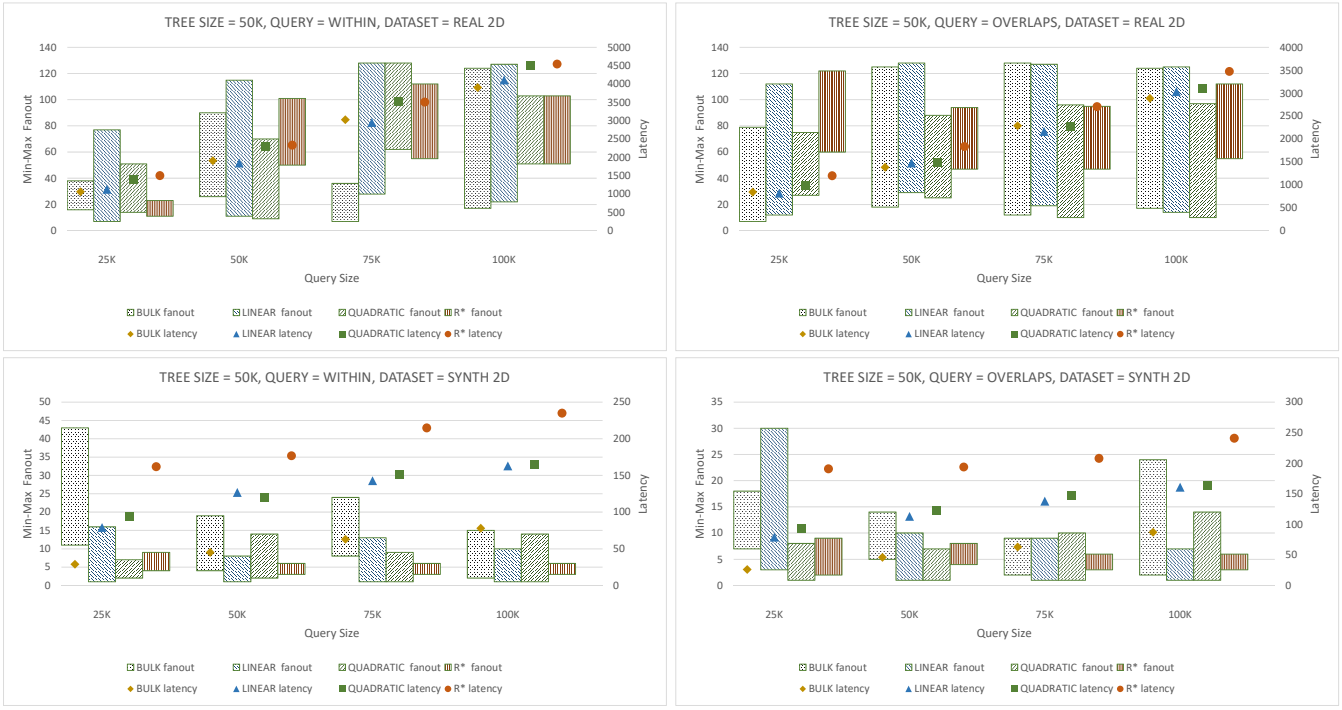

Figure 4: *Within* query for *Bulk* (top left), *Linear* (top right), *Quadratic* (bottom left) and R^* (bottom right) splitting strategies and *Real* dataset.

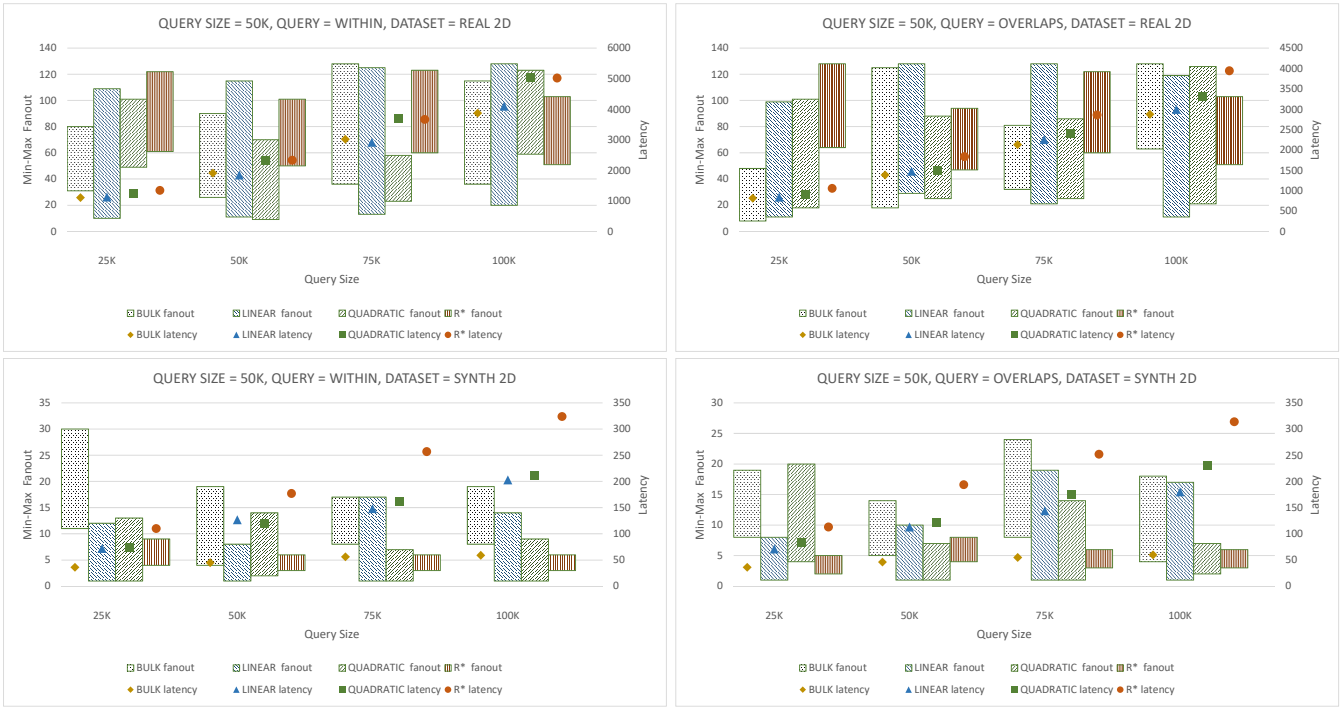

Figure 5: *Overlaps* query for *Bulk* (top left), *Linear* (top right), *Quadratic* (bottom left) and R^* (bottom right) splitting strategies and *Synthetic* dataset.

- to provide dataset visualization that would allow a better interpretation of the optimization results.

REFERENCES

[1] Chuan-Heng Ang and T. C. Tan. 1997. New Linear Node Splitting Algorithm for R-trees. In *Proceedings of the 5th International Symposium on Advances in Spatial Databases (SSD '97)*. Springer-Verlag, London, UK, UK, 339–349. http://dl.acm.org/citation.cfm?id=647225.718938

[2] Manos Athanassoulis and Stratos Idreos. 2016. Design Tradeoffs of Data Access Methods. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 2195–2200. https://doi.org/10.1145/2882903.2912569

[3] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. 2016. Designing Access Methods: The RUM Conjecture. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016*. 461–466. https://doi.org/10.5441/002/edbt.2016.42

[4] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD '90)*. ACM, New York, NY, USA, 322–331. https://doi.org/10.1145/93597.98741

[5] Paul Geoffrey Brown. 2000. *Object-Relational Database Development: A Plumber's Guide with Cdrom*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[6] Douglas Comer. 1979. Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (June 1979), 121–137. https://doi.org/10.1145/356770.356776

[7] Paolo Corti, Stephen Vincent Mather, Thomas J. Kraft, and Borie Park. 2014. *PostGIS Cookbook*. Packt Publishing.

[8] Miguel R. Fornari, Joao Luiz D. Comba, and Cirano Iochpe. 2006. Query Optimizer for Spatial Join Operations. In *Proceedings of the 14th Annual ACM International Symposium on Advances in Geographic Information Systems (GIS '06)*. ACM, New York, NY, USA, 219–226. https://doi.org/10.1145/1183471.1183508

[9] Volker Gaede and Oliver Günther. 1998. Multidimensional Access Methods. *ACM Comput. Surv.* 30, 2 (June 1998), 170–231. https://doi.org/10.1145/280277.280279

[10] Yván J. García, Mario A. Lopez, and Scott T. Leutenegger. 1998. On Optimal Node Splitting for R-trees. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 334–344. http://dl.acm.org/citation.cfm?id=645924.671188

[11] Simon Greener and Siva Ravada. 2013. *Applying and Extending Oracle Spatial*. Packt Publishing.

[12] Oliver Günther, Philippe Picouet, Jean-Marc Saglio, Michel Scholl, and Vincent Oria. 1999. Benchmarking spatial joins a la carte. *International Journal of Geographical Information Science* 13, 7 (1999), 639–655. https://doi.org/10.1080/136588199241049

[13] Christophe Gurret, Yannis Manolopoulos, Apostolos Papadopoulos, and Philippe Rigaux. 1999. The BASIS System: A Benchmarking Approach for Spatial Index Structures. In *Proceedings of the International Workshop on Spatio-Temporal Database Management (STDBM '99)*. Springer-Verlag, London, UK, UK, 152–170. http://dl.acm.org/citation.cfm?id=646518.694064

[14] Antonin Guttman. 1984. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD '84)*. ACM, New York, NY, USA, 47–57. https://doi.org/10.1145/602259.602266

[15] Erik G. Hoel and Hanan Samet. 1995. Benchmarking Spatial Join Operations with Spatial Output. In *Proceedings of the 21th International Conference on Very Large Data Bases (VLDB '95)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 606–618. http://dl.acm.org/citation.cfm?id=645921.673135

[16] James Kennedy and Russell Eberhart. 1995. Particle swarm Optimization. In *Proceedings of IEEE International Conference on Neural Networks*, Vol. 4. 1942 – 1948 vol.4. https://doi.org/10.1109/ICNN.1995.488968

[17] Marcel Kornacker, Mehul Shah, and Joseph M. Hellerstein. 1999. *An Analysis Framework for Access Methods*. Technical Report UCB/CSD-99-1051. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/1999/5520.html

[18] Ravikanth V. Kothuri, Albert Godfrind, and Euro Beinat. 2007. *Pro Oracle Spatial for Oracle Database 11G (Expert's Voice in Oracle)*. Apress, Berkely, CA, USA.

[19] Hans-Peter Kriegel and Michael Schiwietz. 1990. Performance Comparison of Point and Spatial Access Methods. In *Proceedings of the First Symposium on Design and Implementation of Large Spatial Databases (SSD '89)*. Springer-Verlag, London, UK, UK, 89–114. http://dl.acm.org/citation.cfm?id=647221.718589

[20] Scott T. Leutenegger, J. M. Edgington, and Mario A. Lopez. 1997. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *Proceedings of the Thirteenth International Conference on Data Engineering (ICDE '97)*. IEEE Computer Society, Washington, DC, USA, 497–506. http://dl.acm.org/citation.cfm?id=645482.653437

[21] Yan Liu, Jinyun Fang, and Chengde Han. 2009. A new R-tree node splitting algorithm using MBR partition policy. In *Proceedings 17th International Conference on Geoinformatics*. IEEE Computer Society, 1–6. https://doi.org/10.1109/GEOINFORMATICS.2009.5293260

[22] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadopoulos, and Yannis Theodoridis. 2005. *R-Trees: Theory and Applications*. Springer Publishing Company, Incorporated.

[23] Regina O. Obe and Leo S. Hsu. 2015. *PostGIS in Action* (2nd ed.). Manning Publications Co., Greenwich, CT, USA.

[24] Philippe Rigaux, Michel Scholl, and Agnès Voisard. 2002. *Spatial databases - with applications to GIS*. Elsevier.

[25] Karl Schnaitter and Neoklis Polyzotis. 2009. A Benchmark for Online Index Selection. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*. 1701–1708. https://doi.org/10.1109/ICDE.2009.166

[26] Baron Schwartz, Peter Zaitsev, and Vadim Tkachenko. 2012. *High Performance MySQL: Optimization, Backups, and Replication* (3rd ed.). O'Reilly Media, Inc.