To Sort or not to Sort: The Evaluation of R-Tree and B^+-Tree in Transactional Environment with Ordered Result Set Requirement [*]

© Pavel Fedotovsky George Erokhin Kirill Cherednik Kirill Smirnov
George Chernishev

Saint-Petersburg University
{pavel.v.fedotovsky, george.erokhin, chernishev}@gmail.com,
{kirill.cherednik, kirill.k.smirnov}@math.spbu.ru

Abstract

In this paper we consider multidimensional indexing with the additional constraint of lexicographical ordering. In order to deal with this problem we discuss two well-known tree data structures: R-Tree and B-Tree. We study the problem in the transactional environment with read committed isolation level. To evaluate these approaches we had implemented these structures (modified GiST ensures concurrency) and provide extensive experiments.

1 Introduction

In this paper we consider the problem of multidimensional indexing with one additional constraint — the lexicographical ordering of the resultset. Effective multidimensional indexing is rather old and well-explored topic, however, one can't say that the problem is solved. New approaches continue to emerge. The addition of the ordering requirement further drives this problem into the domain of research activity.

Effective solutions for the problem of multidimensional indexing are needed for geospatial data, CAD systems, multimedia data and also of use for OLAP data.

There are two main approaches for multidimensional indexing: tree-based and hash-based. The former are R-Tree, KDB tree, Octree, X-Tree and many others. The latter are mainly used for nearest neighboor and similarity query evaluation.

We are mainly interested in R-Tree because of it's popularity in commercial DBMS systems [4]: PostgreSQL, Oracle, Informix, SQLite and MySQL use this approach. This interest proves, that despite being rather old (more than 25 years), R-Tree still may be called industrial-strength technology. Moreover, until recently R-Tree was the only one method of multidimensional indexing in PostgreSQL[1].

This work was inspired by participation in ACM SIGMOD Contest 2012. This problem was provided by the contest organizers, as well as benchmarks and example Berkeley DB-based implementation. Our team participated in this contest and was ranked 5th on public tests[2].

The problem is formulated as follows: given a n-dimensional space and queries in transactional environment, what kind of data structure should we use for optimal performance?

In order to solve this problem we implemented a prototype of multidimensional transactional index. This index works within read committed isolation level. Our prototype contains both B^+-Tree and R-Tree built around GiST model.

The contribution of this paper is following:

- The validation of our prototypes by comparison with industrial-strength databases: Berkeley DB and PostgreSQL.

- Experimental study of influence of workload parameters on performance of these two structures. These workload parameters include query window size and others.

The rest of this paper is organized as follows. In the next section we provide detailed specification of the task, describe queries and data. Then, in the section 3 we describe two alternative approaches and survey related works. Section 4 contains overview of our system. In the section 5 we provide evaluations and comparisons with PostgreSQL and Berkeley DB.

2 The Task

The task offered at the contest was to build a multidimensional high-throughput in-memory indexing system. The index should support concurrent access by many threads and work within read committed isolation level. The index resides in-memory and no crash-recovery component is required.

[*] This work is partially supported by Russian Foundation for Basic Research grant 12-07-31050.

Proceedings of the Ninth Spring Researcher's Colloquium on Database and Information Systems, Kazan, Russia, 2013

[1] `http://www.postgresql.org/docs/9.2/static/spgist.html`

[2] `http://wwwdb.inf.tu-dresden.de/sigmod2012contest/leaderboard/`

2.1 Queries

There are several possible types of queries:

- Point queries: insert, update, delete and select.
- Range queries — they select a subset of data and the result should be sorted. This type of query is defined by a conjunction of attribute predicates. The individual predicates may be not only be intervals or points, but also a wildcards.

The distribution of query types is described in the specification and it can be tuned.

Another important aspect to consider is the admissible amount of operations per transaction. It is specified, that there are no more than few hundred retrieved points per transaction. In particular, the original task states that no more that 200 points are touched by any transaction. This number is justified by the fact that OLTP transactions are very light-weight. For examle, the heaviest transaction in TPC-C reads about 200 records [12].

2.2 Data And Workloads

The task statement specifies several datatypes:INT(4), INT(8) and VARCHAR(512). However, in this work, we had to drop VARCHAR (see section 5 for details). The key consists of several attributes of these datatypes. The payload is represented by a sequence of bytes.

The data may come in one of several types of distributions: normal, uniform and zipf (each is applied to coordinate independently). In our tests we used only uniform one.

Duplicate keys are allowed, we refer the reader to the web site for the detailed handling description.

In our experiments we heavily rely upon workloads and benchmark driver provided by organizers. These workloads are essentially synthetic datasets. We don't reuse workloads used during the contest, instead we use the provided framework to define our own.

Thorough task specification can be found here[3].

3 Related Work And Architectural Alternatives

In order to solve this problem two architectural approaches may be used. The first one is to use B^+-Tree and concatenate the values of individual coordinates into the composite key. The B^+-Tree [13] is the balanced data structure, which contains values in the leaf nodes while inner nodes contain pointers and intervals. These intervals define the unique path to the leaf.

The strong points of this approach are:

- The overall simplicity of this data structure and general easiness for implementation.
- The abudance of concurrency control mechanisms for this kind of tree [13].
- It is possible to tune one, a lot of cache-conscious modifications exist.
- No need to sort, because keys are already stored in the right order.

Let's review the last item. Suppose that we have a three dimensional index and a query: $(1, 2, *)$. In order to evaluate it, we have to find the first entry with prefix "1|2|" and then sequentially scan the tree until prefix mismatch.

However one can name weak points:

- We have to pack and unpack the keys with each comparison.
- Queries containing interval predicates are harder to process.
- This tree may perform poorly with wildcard queries.

The first one is the minor drawback, its cost may be negligible. However, the second and the third are more formidable ones.

The intervals inside attributes can be processed in the same manner as above, but additional checks are needed. This results in additional complexity of the implementation.

Regarding the third item, consider query $(1, *, 3)$. In order to evaluate it, we have to find the key starting with a prefix "1", then we have to iterate through all values which have it. It will require a lot more of comparisons, and what is more important, we will be forced to discard a lot of value in the middle. Consider the following leaf level:

$$1|2|3|, 1|2|4|, 1|2|4|, ..., 1|2|4|, 1|3|3|.$$

In this situation we will need only two values: $1|2|3|$ and $1|3|3|$. But we would be forced to iterate through all these values and discard them.The situation becomes grave when we have wilcard condition in the first attribute: $(*, 2, 3)$. In this case we have to scan the whole index.

R-Tree is the specialized data structure proposed first by Antonin Guttman in [6]. This study prompted a wave of research papers and one can say that it gave birth to the new area of research. This research related to development of the new R-Tree variants [4, 8, 11], niche approaches [10, 11], split techniques [2, 3, 5], concurrency techniques [7, 9] etc. The study [10] states that there is more than 100 variants of R-Trees.

R-Tree can be thought of as an extension of B-Tree for multidimensional indexing. It shares some concepts:

- Data are kept in the leaves, too.
- This data structure is also balanced.
- Inner nodes keep bounding boxes, which may be thought as the generalization of intervals.

The main differences are:

- There may be more than one path to the key. This is the result of bounding box intersection permission.
- Node split is unambiguous, determining the optimal node split is a very hard problem.

[3] http://wwwdb.inf.tu-dresden.de/sigmod2012contest/

- No link to sibling leaves for easy range query execution.

GiST (Generalized Search Tree) [9] is a “template” index structure which supports extensible set of queries and datatypes. This index can be parametrized by a variety of data structures.

Unlike B^+-Tree based one, this approach would require sorting of the results. This is a significant drawback which may negatively impact performance. The goal of this paper is to evaluate, which of these approaches is better. Intuitively one can say that the outcome should depend on the query selectivity.

4 System Overview

Our system follows classical design guidelines and contains several components:

- A tree data structure. Currently implemented as B^+-Tree and R-Tree. R-Tree is based upon GiST [7], a popular template index structure including concurrency control techniques. This model allows to extend with the means of concurrent access almost any tree conforming to certain requirements. This is a widespread approach and it is used, for example, in PostgreSQL.

- Concurrency control. We used mechanism adapted from [9] with locks, latches and Node Sequence Numbers. Also we provided deadlock resolution mechanism. Eventually, we ensure the read committed isolation level. However currently our prototype lacks logging and recovery features.

- Memory manager. It is a well-know fact that a standard memory manager can’t provide optimal performance for the whole range of applications and sometimes it is desirable to find or implement a specifically-tailored one. Our memory manager is essentially a wrapper which intercepts new and delete calls to make use a pool of free blocks.

- Sorting of the results. In order to solve the problem one must present lexicographically sorted results. While B^+-Tree provides already ordered results, R-Tree does not. Our R-Tree implementation sorts the results via merge-sort (we keep sorted data inside boxes).

- Deletion of records. In our implementation we don’t delete records, instead, we mark them as “deleted” and take this into account during the processing.

5 Validation and Experiments

5.1 Validation

We validated our implementation in two ways. First, we used public unit-tests supplied by the contest organizers. These unit-tests ensured correctness of an isolation level (read committed) implementation and several other implementation issues. We also extended basic set of test cases with new ones. Then, our implementation participated in the contest [1].

5.2 PostgreSQL validation and tuning

We also compared our implementation with PostgreSQL v9.1 database system. This step was needed to check the relative level of achieved performance and general transferability of results. We implemented a simple wrapper application which directed queries to PostgreSQL. PostgreSQL uses a disk-based GiST index, while our prototype is an in-memory one. Also, our prototype lacks a logging and recovery component. Thus, in order to conduct fair tests we had to simulate in-memory index in PostgreSQL.

To completely eliminate slow disk-related operations we placed database cluster on tmpfs. This way we can be sure that every operation PostgreSQL performs (logging, committing, buffers flushing, etc) does not involve interactions with a hard drive.

Other important implementation aspects included:

- Wrapper connection pooling. We used a pool of connections inside our wrapper to eliminate the cost of connection creation every time a transaction is executed.

- We parametrized GiST with cube data structure.

- To eliminate overheads related to durability we turned off: fsync, full page writes and synchronous commit. Checkpoint segments setting was left intact.

- We were forced to abandon string datatype due to PostgreSQL cube restrictions (only float parameters supported).

- PostgreSQL runs in read committed isolation level by default.

Unfortunately, due to several reasons, we were not able to completely approach the performance of our system. First, unlike BDB, PostgreSQL needs to maintain not only the index, but also a table. Second, calls to PostgreSQL via connections are less effective than the direct function calls. The last issue is the security checks which were also left intact.

5.3 Hardware and software setup

For the first group of experiments (comparison with PostgreSQL and Berkeley DB) used the following hardware and software setup:

- Intel Core i7-2630QM, 2.00 GHz, Hyper-Threading Enabled, L1 Cache 64KB, L2 Cache 256 (per core), L3 Cache 6MB, 6GB RAM

- x86_64 GNU/Linux, kernel 3.5.0-21, gcc 4.7.2

- PostgreSQL 9.1.7

The second group used the more performing one:

- Hardware: 2 x Intel Xeon CPU E5-2660 0 @ 2.20GHz, 64GB RAM, MB S2600GZ

- Software: Linux Ubuntu 3.2.0-29-generic x86_64, GCC 4.6.3

5.4 Comparison with PostgreSQL and Berkeley DB

In this section we provide a comparison of our prototypes with industrial strength systems. The wrapper for Berkeley DB was provided by the organizers, PostgreSQL wrapper was developed by the authors (it's architecture was described earlier). We compare the performance varing the number of dimension and use single 64MB index. The query type distribution is the same as in the original contest task, uniformly distributed data was used.

We can see:

- Our prototypes are comparable to industrial ones in terms of overall performance.
- The solution which uses R-Tree significantly differs from B^+-Tree in terms of performance. This difference has prompted us into further investigation, which resulted in this paper.

5.5 Experimental Evaluation

The goal of this paper is to evaluate, what is better: to use R-Tree and to sort or not to sort with B^+-Tree, but risk excess comparisons.

In order to solve this problem we had conducted a series of experiments. In these experiments we evaluate the performance of two systems, while varing the query selectivity. We separately consider the following dimensions: 2, 4, 6, 8. We had considered indexes of two sizes: 64 and 512 MB, uniform data distribution. We concentrate on the most interesting query type, which present in the original contest workload: a range query without wildcard predicates. These experiments were conducted using our prototypes, which we had described in the previous section. The reason of this switch is the time it takes to construct an index by PostgreSQL DBMS and also the query plan problem. The plans which are generated by the optimizer are essentially the following: at first, perform index scan (e.g. read all R-Tree boxes), then sort the results. It is impossible to push down sorting in PostgreSQL because it's GiST selection method doesn't uses merge-sort. This is a critical drawback, because in our task we select at most 200 entries. Thus, our prototype can read only a part of the data and don't sort all the content of the touched boxes. The query plan problem is not an issue in BDB, because of the simplicity of BDB and the fact that B^+-Tree is already sorted.

The results are presented on Figures 3-6.

Note the double logarithmic scales, which we used in order to illustrate our finds. They are the following:

- The throughput of the system depends on a query selectivity. This dependence can be described by the power law:

$$P = a * S^b,$$

  where P denotes the throughput, S — query selectivity, a and b are parameters. The graphs show this kind of dependency by the straight line. This approximately linear dependency persists in all considered dimension sizes.

Tree type	2	4	6	8
R-Tree (64MB)	-0.43	-0.22	-0.10	-0.01
R-Tree (512MB)	-0.61	-0.25	-0.11	-0.01
B^+-Tree (64MB)	0.50	0.69	0.62	0.55
B^+-Tree (512MB)	0.49	0.70	0.60	0.40

Table 1: Parameter values for R-Tree and B^+-Tree

- The considered query type affects the performance of the systems in the following way: the performance of R-Tree degrades as the value of query selectivity decreases, while at the same time B^+-Tree performance increases.
- As the number of dimensions increases, the exponent b changes in the way shown in the Table 1. Increasing the dimensionality leads to b decrease in case of R-Tree, i.e. having more dimensions lowers impact of query selectivity. There is no manifested trend in B^+-Tree behaviour.
- The following hypothesis can be advanced: the exponent in power-law does not depend on size of the index, only on dimensionality. To prove this hypothesis more thoroughful investigation is needed.
- There is no simple way to determine intersection point of R-Tree and B-Tree, it depends on number of dimensions and index size.

6 Conclusions

In this paper we have considered the problem of multidimensional point indexing and under condition of additional restriction: ordering the results. We have experimentally evaluated two data structures — R-Tree and B^+-Tree on uniformly distributed data. The experiments allowed us to establish the impact of the query selectivity on system performance as power function. Also we examined the dependency of power-law parameters on dimension. As a future work we will provide more empirical evidence to the hypothesis of independance of power-law exponent on index size. Recommendation for B-Tree and R-Tree user: unfortunately, we were not able to find an easy way to calculate intersection point, so workloads should be evaluated ad hoc.

7 Acknowledgements

We would like to thank organizers of ACM SIGMOD Programming Contest'12 for providing a benchmark, data generator, unit tests and Berkeley DB wrapper implementation. This work is partially supported by Russian Foundation for Basic Research grant 12-07-31050.

References

[1] ACM SIGMOD Programming Contest'12. http://wwwdb.inf.tu-dresden.de/sigmod2012contest/. Last accessed: 11/09/2012.

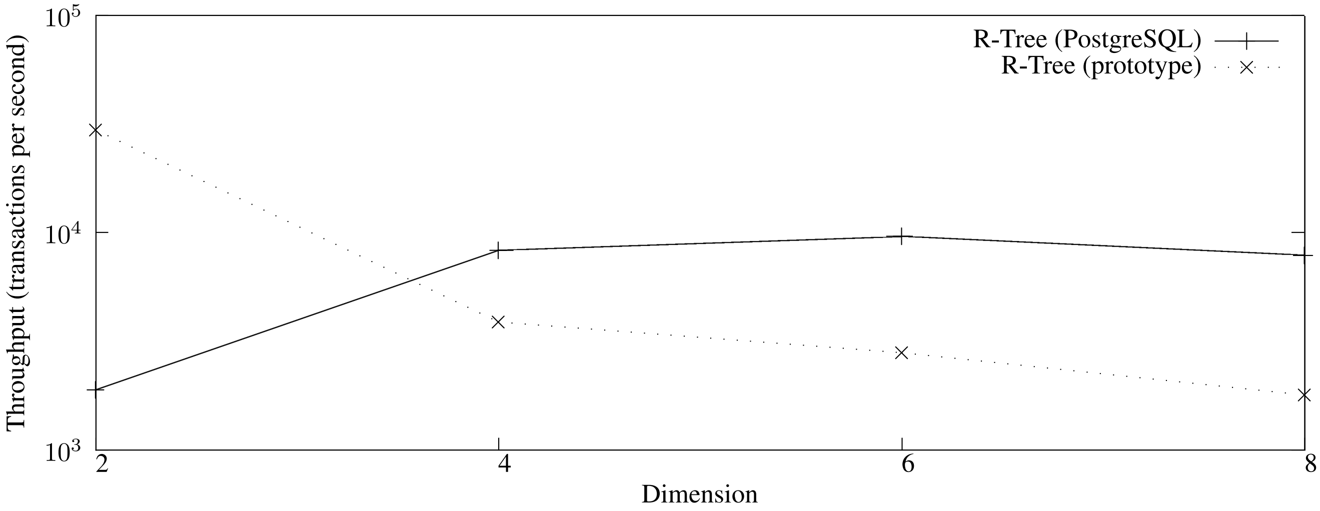

Figure 1: Performance of PostgreSQL and our prototype (R-Tree).

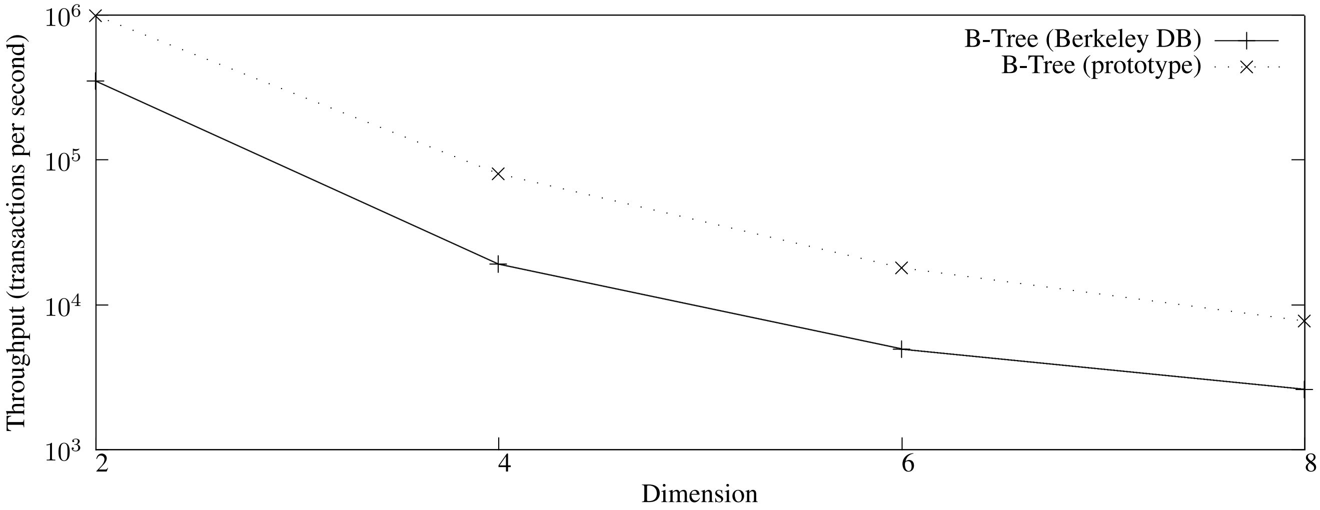

Figure 2: Performance of Berkeley DB and our prototype (B^+-Tree).

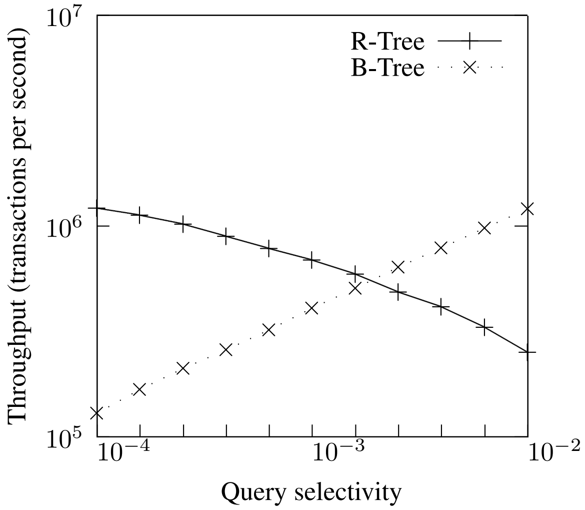

Figure 3: Performance of R-Tree and B-tree indexes, 64MB, 2 dimensions (first hardware setup).

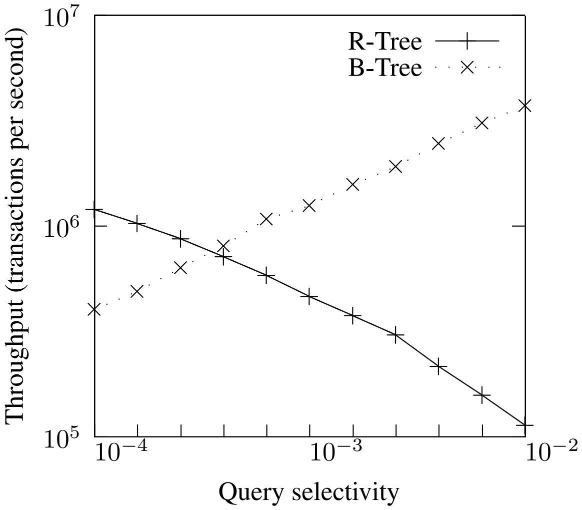

Figure 4: Performance of R-Tree and B-tree indexes, 512MB, 2 dimensions (second hardware setup).

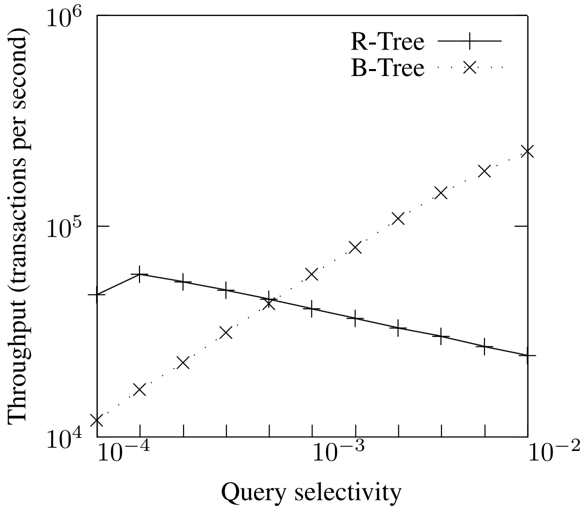

Figure 5: Performance of R-Tree and B-tree indexes, 64MB, 4 dimensions (first hardware setup).

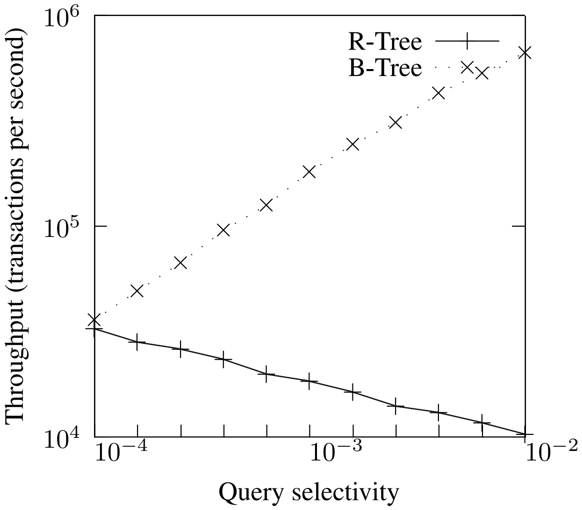

Figure 6: Performance of R-Tree and B-tree indexes, 512MB, 4 dimensions (second hardware setup).

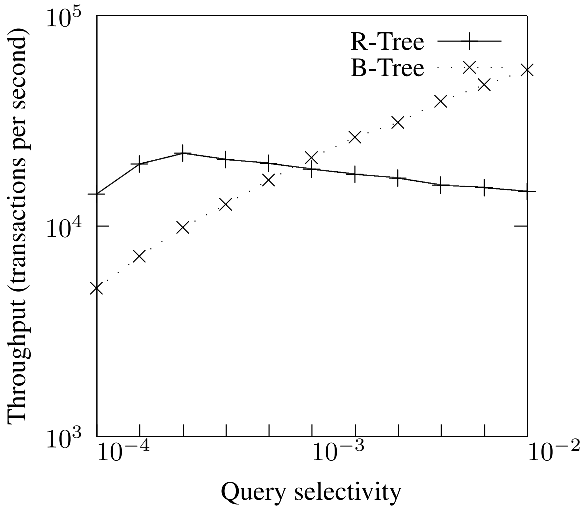

Figure 7: Performance of R-Tree and B-tree indexes, 64MB, 6 dimensions (first hardware setup).

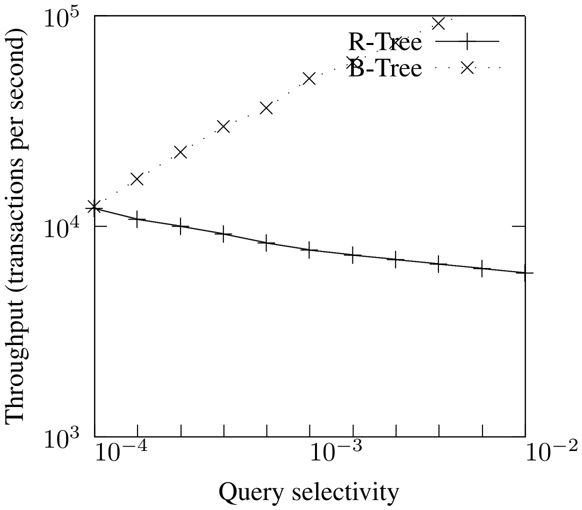

Figure 8: Performance of R-Tree and B-tree indexes, 512MB, 6 dimensions (second hardware setup).

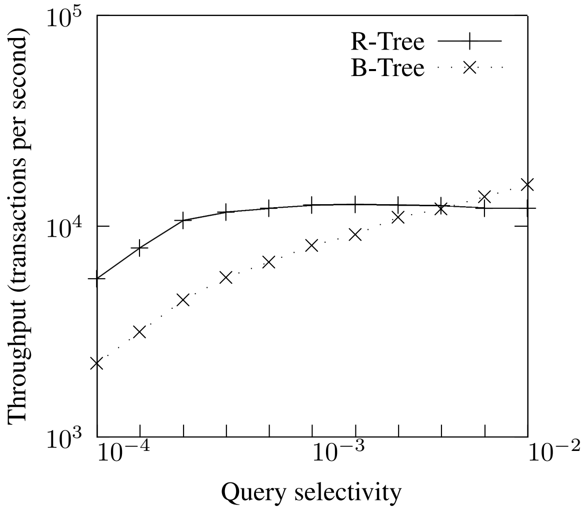

Figure 9: Performance of R-Tree and B-tree indexes, 64MB, 8 dimensions (first hardware setup).

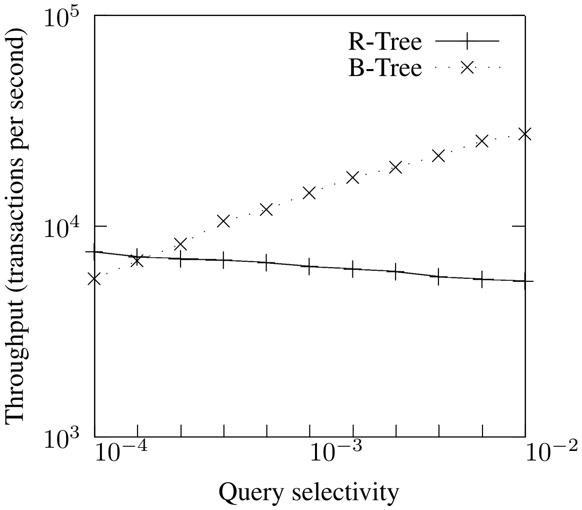

Figure 10: Performance of R-Tree and B-tree indexes, 512MB, 8 dimensions (second hardware setup).

[2] Amer F. Al-Badarneh, Qussai Yaseen, and Ismail Hmeidi. A new enhancement to the R-tree node splitting. *J. Inf. Sci.*, 36(1):3–18, feb 2010.

[3] C. Ang and T. Tan. New linear node splitting algorithm for R-trees. In Michel Scholl and Agnès Voisard, editors, *Advances in Spatial Databases*, volume 1262 of *Lecture Notes in Computer Science*, pages 337–349. Springer Berlin / Heidelberg, 1997. 10.1007/3-540-63238-7_38.

[4] Norbert Beckmann and Bernhard Seeger. A revised R*-tree in comparison with related index structures. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, pages 799–812, New York, NY, USA, 2009. ACM.

[5] Sotiris Brakatsoulas, Dieter Pfoser, and Yannis Theodoridis. Revisiting R-Tree Construction Principles. In *Proceedings of the 6th East European Conference on Advances in Databases and Information Systems*, ADBIS '02, pages 149–162, London, UK, UK, 2002. Springer-Verlag.

[6] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, June 1984.

[7] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized Search Trees for Database Systems. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 562–573, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[8] Ibrahim Kamel and Christos Faloutsos. Hilbert R-tree: An Improved R-tree using Fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 500–509, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[9] Marcel Kornacker, C. Mohan, and Joseph M. Hellerstein. Concurrency and recovery in generalized search trees. *SIGMOD Rec.*, 26(2):62–72, June 1997.

[10] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadopoulos, and Y. Theodoridis. *R-Trees: Theory and Applications (Advanced Information and Knowledge Processing)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[11] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The R^+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proceedings of the 13th International Conference on Very Large Data Bases*, VLDB '87, pages 507–518, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[12] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[13] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.