R-tree re-evaluation effort: a report

Kirill Smirnov, George Chernishev, Pavel Fedotovsky, George Erokhin, and Kirill Cherednik

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

Abstract. In this report we describe our scalability-related experiments with the transactional R-Tree prototype on a server-class hardware. We describe here our algorithm for concurrency control and transactional isolation, the prototype and the evaluation environment. Also the motivation and a short R-Tree technology survey is presented. This report can be considered as the supplementary to the paper [1], which was shrunk in order to fit into the space constraints.

Keywords: threads, scalability, databases, multidimensional indexing, in-memory index, R-Tree, GiST, experimental evaluation

1 Motivation

Practical evaluations of concurrent implementations of the R-Tree were already performed in the past. However they were performed a long time ago (10-15 years) and thus do not comply with the state of the art. The eras of multi-core processors, cloud computing and main-memory systems require additional evaluations. Essentially, there are four main reasons for this reevaluation:

– New hardware. To the best of our knowledge, newest experiments which were conducted with concurrent implementation of the R-Tree used the hardware dated to 2003 year [2] or even earlier. Also, previous studies used SPARC hardware [2, 3], which is quite different [4] from the contemporary mainstream hardware.
– In-memory. Large portion of the previous studies used disk-based prototypes [3]. In the past years there is a trend to move away from disk-based systems to in-memory ones. This type of systems is becoming increasingly popular due to the vast amounts of cheap memory being available and it is believed that these systems will dominate the OLTP market soon [5, 6].
– Small dataset sizes were used in the past. To the best of our knowledge, we use the biggest one up to date. For example, the [2] study featured the data size of 1 MB (tree size was about 50 MBs). Exact raw data size is unclear

* This work is partially supported by Russian Foundation for Basic Research grant 12-07-31050
due to complicated storage scheme. Another study used 1.2 millions of three
rectangles. In our experiments we consider workloads up to 1GB of raw data
(our tree was tens of GBs). We think that this is the largests dataset used
in the experiments with the concurrent implementation of the R-Tree.
- Scalability issues. To the best of our knowledge, our study is the first evalu-
ation which considers the scalability metrics of the concurrent R-Tree imple-
mentation. We do not know any other study which considers this problem
in context of the R-Tree. The era of cloud computing and big data make the
problem of scalability the most acute.

The goal of this study is to present our prototype and to provide some initial
experiments regarding the problem.

2 Introduction

According to [8], an index is defined as a “data structure plus a method of
arranging the data tuples in the table (or other kind of collection object) being
indexed”. An index is usually used to reduce query evaluation time, depending
on its type. For example, B-tree allows to lower the computational complexity
of search queries from linear to logarithmic. However, these benefits come with
a price: one have to maintain the structure and pay upkeep costs. There are two
popular ways to construct an index: using a hash table and using a search tree.
This work considers the latter approach.

Multidimensional indexing differs from the standard one in the following way:
an indexing key is a composite structure which contains several attributes (i.e.
a point in multidimensional space). This allows for additional types of queries
such as [9]: range, nearest neighbor, spatial and directional.

Two-dimensional indices are widely used in geographic information systems
to efficiently evaluate queries like: "find all cities that are in the rectangle" (range
query) or "find five nearest cities for a given point" (nearest neighbor query).

Efficient multidimensional indexing is required for handling geospatial and
multimedia data, CAD systems, also of some use for OLAP databases.

R-Tree is a de facto standard of the database industry for multidimensional
indexing: PostgreSQL, Oracle, Informix, SQLite and MySQL follow this ap-
proach [10]. However, the problem of efficient multidimensional indexing is very
far from being solved. Moreover, the problem becomes challenging when one
considers it in a transactional environment. This means that, inter alia, a pro-
gramer should provide [11]:

- A concurrency control mechanism to ensure the integrity of a data structure
during multithreaded access. The index must preserve its internal structure,
i.e. broken links or dangling pointers must not appear.
- Some guarantees for the outcome of conflicting operations. These guarantees
  are specific to the area of databases and are defined by the specified isolation
  level.
Performance and correctness of parallel access are not the only important characteristics. This access should also be scalable, e.g. the performance of such system should benefit from the addition of new processing elements (processors, cores).

The goal of this study is to explore the scalability issues of such system and to study the effect of adding more cores in a variety of scenarios. We consider a multidimensional indexing system in a transactional environment.

This work uses a prototype implementation of transactional, in-memory, multidimensional indexing system based on R-Tree and the GiST model [12]. This system is the follow-up of the ACM SIGMOD Contest 2012, where initial version was created. It was ranked 5th on the public (preliminary) tests\(^1\).

The contribution of this report is the following:

– We provide a short survey of the related R-Tree technology.
– A modification of the GiST transactional isolation algorithm for the read committed level is proposed.
– Details regarding the experimental evaluation of scalability of R-Tree-based indexing system are presented.

This report can be considered as the supplementary material to the paper [1], which was shrunk in order to fit into the space constraints.

## 3 R-Tree: a Short Survey

One of the most popular data structures used for indexing multidimensional data is R-tree [9]. According to [13], R-tree is a tree data structure, defined by a pair \((m, M)\) with the following properties:

– Each leaf node (unless it is the root) can hold \([m, M]\) entries, where \(m \leq M/2\). Each entry is represented by a pair \((\text{mbr}, \text{oid})\), where \text{mbr} is the minimum bounding rectangle (MBR) that spatially contains the object and \text{oid} is the object’s identifier.
– The number of entries that each internal node can store is in \([m, M]\). Each entry is represented by a pair \((\text{mbr}, p)\), where \(p\) is a pointer to a child of the node and \text{mbr} is the MBR that spatially contains the MBRs contained in this child.
– The minimum allowed number of entries in the root node is 2, unless it is a leaf (in this case, it may contain zero or a single entry).
– R-Tree is balanced — all leaves of the R-tree reside at the same level.

The example data and the corresponding R-Tree are presented at Fig.1. The data consist of boxes marked E-O and boxes A-D denote the MBRs. The latter are depicted by dotted rectangles.

One may think of an R-Tree as a generalization of \(B^+\)-Tree:

Data are also kept in the leaves.
This data structure is also balanced, all leaf nodes are at the same height.
Inner nodes keep bounding boxes, which can be considered as a generalization of intervals which guide the search in $B^+$-Tree.

However, there are several differences:

- It might be necessary to explore more than one path from the root to a leaf in order to locate a key. This is the result of the MBR intersection allowance, which leads to more complex search and traversal algorithms than those of $B^+$-Tree.
- A node split is ambiguous, determining the optimal node split is a computationally hard problem. A “bad” split may lead to a serious degradation of the performance.
- R-Tree does not contain links to sibling leaves for an easy range query execution.

R-Tree can be used to efficiently evaluate queries of the following types:

- Point queries: insert, update, delete. These are essentially manipulations which involve one data record;
- Range query — a query involving all points (records) which fit into the specified multidimensional rectangle. This is the core query type for multidimensional indexing [9]. In this paper we consider read-only range queries.
The R-Tree was first proposed by Antonin Guttman in 1984 [14]. This paper was followed by a steady stream of works which further extended this approach. Overall, in twenty years there were proposed almost hundred variants of R-Tree [13]. The most notable variants are $R^+$-Tree [15], $R^*$-Tree [16], Hilbert R-Tree [17].

Not only new variants were developed, but also related studies were conducted, for example the bulk loading problem was considered (how to load a big volume of data into R-Tree) [18–20], the problem of a node split (how to produce nodes optimal for further searches) [21–23] and many others. For complete reference see [13].

One of the well-known mechanisms for concurrency control and transactional isolation in R-Trees was devised around 1995 [12, 11]. This development paved the way for applications of this structure in transactional systems.

Later, the industrial adoption followed, for example [24] (the complete list of systems was given in the introduction). According to [9], R-Tree succeeded because of its resemblance to $B^+$-Tree. Nowadays one may say that R-Tree became a de facto industrial standard of multidimensional indexing. However, despite the data structure is rather old (more than 25 years), research is still active [10, 25, 26]. More details regarding the R-Tree research area (and works up to 2005) are covered in [13].

4 Concurrency Control, Transactional Isolation and Related Concepts

An indexing system should provide a concurrency control to allow multiple threads to access the data structure (an index). A great deal of care should be taken of it, because threads may be caught in the middle of a reorganization operation (for example, a node split).

Transactional database systems should conform to the ACID properties [27]. The most important ones for us are the first three: atomicity, consistency and isolation. Ensuring these properties is the very hard scientific and technological task.

There are several degrees of such conformity, ranging from the strictest “serializable” to the weakest “read uncommitted” [27]. These levels are called isolation levels and they define behavior of a transactional system. They permit or disallow several scenarios of processing of conflicting operations. The difference between these scenarios is the allowed anomalies. A database administrator may choose the isolation level depending on the specifics of an application which uses a database. One of the most commonly used is the level called “read committed” and this level is considered in this paper. It is the default isolation level of many industrial DBMS.

Summarizing this section, concurrency control and transactional isolation protocols are the mechanisms to manage the transaction processing. They heavily rely on special techniques like locks, latches etc [27]. We use one of these
protocols designed for the GiST template (discussed next) [11] and tailor it for the considered isolation level “read committed”.

4.1 Concurrency Control and Transactional Isolation

GiST provides concurrency control as well as transactional isolation for supported data structures. Concurrency control for GiST is well-described in the original paper [11], our implementation follows that template. Transactional isolation is handled differently: original paper provides repeatable read isolation level which is more strict than read committed level. Therefore it is a subject for adaptation.

In order to ensure read committed isolation level, one has to prevent the following anomalies: a dirty read and a lost update [27]. So, we modify the original method in the following way. First of all, predicate locking is not used, secondly, each record holds the id of current writing transaction as well as operation being performed by this transaction. There are five states:

- **ProcessInsert** — the record is being inserted.
- **ProcessDelete** — the record is being deleted.
- **ProcessUpdate** — the record is being updated (only payload is changing, but key remains the same).
- **Deleted** — the record is being deleted.
- **Valid** — the record is not being changed currently.

Deletion algorithm is only logical, i.e. only “record deleted” flag is set. Let’s describe how basic operations are performed:

- **Select.** All required records are retrieved using search tree. Records are processed differently depending on state:
  - **ProcessInsert** — the record is skipped as the transaction that inserted it has not been committed yet.
  - **ProcessDelete** — the record is treated as “alive” since the record the transaction that deleting it has not been committed or reverted yet (otherwise state would be Deleted or Valid respectively).
  - **ProcessUpdate** — an old value of the record is returned.
  - **Deleted** — the record was deleted so it is skipped (see ProcessInsert).
  - **Valid** — the record is passed to user as there is no transaction currently modifying it.

- **Insert.**
  - The record is marked with “ProcessInsert” label and current transaction id.
  - The record is inserted into the tree data structure.

- **Update/Delete.**
  - Transaction searches for record which needs to be updated/deleted.
  - Record modification (updating state and value).

Why this algorithm prevents anomalies and ensures the stated isolation level (read committed):
– **Lost Update.** Occurs when two transactions overwrite the same record. It is eliminated as follows: every time transaction updates a record it requests a lock that ensures the record being updated only by one transaction.

– **Dirty Read.** Occurs when a transaction is allowed to read data that has been modified by another running transaction and have not been committed yet. To avoid this anomaly we use the suggested label mechanism. For each record that is being read its state is analysed. If the record is being updated the old value that is already committed would be returned.

### 5 Experiments

#### 5.1 Related work

There are a lot of aspects to consider in the experimental evaluation of transaction processing systems for multidimensional data. Aside from the spatial aspects there are the transaction-specific aspects which are also should be taken into account. Lets consider them.

The first aspect is how to compose the workload, e.g. what is a transaction set and what is an individual transaction. Lets review some of the existing approaches:

– “Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems” [28].
  
  “We ran each concurrency control scheme with following workloads: 100% search, 100% insert, 50% insert + 50% delete and a mixture of search, insert and delete with varying update ratio. We do not show the graph of 100% insert due to lack of space. The graph is very similar to that of 50% insert + 50% delete.”

– “Efficient Concurrency Control in Multidimensional Access Methods” [7].
  
  For the most of experiments the following holds:
  “transaction size=10 (the number of operations per transaction), write probability=0.2 (the fraction of operations in a transaction that are writes (i.e. inserts))”

  However authors also vary this number in one experiment (see parameter list below).

– “High-Concurrency Locking in R-Trees” [3]. Unfortunately, it is not clear what kind of transaction specification was used by authors. It looks like they used transactions consisting of one operation.

– “High-Performance Concurrency Control” [29].
  
  “We focus on short update transactions which are common for OLTP workloads. The workload consists of a single transaction type that performs R reads and W writes against a table of N records with a unique key.”

  “There are two transaction types running under Read Committed isolation: the update transaction performs 10 reads and 2 writes (R=10 and W=2), while the read-only transaction performs 10 reads (R=10 and W=0).”
“The benchmark runs a random mix of seven short transactions; each transaction performs less than 5 operations on average. 80% of the transactions executed only query the database, while 16% update, 2% insert, and 2% delete items.”

– “Performance Evaluation of Main-Memory R-tree Variants” [2]. This is the most relevant paper to our study.

“Node fanout, Index height and Index size for different node sizes (data size=1M, uni-form dist.).”

“We use two region queries, where the regions are specified with square windows whose centers are distributed uniformly within the 40km × 40km region. The window sizes are 126m × 126m and 400m × 400m and the resulting selectivity values are 0.001% and 0.01%, respectively.”

“Updates are performed as a sequence of random moves. Each move deletes an entry at the coordinates < x, y > and inserts it into a new position at < x − 30 ∗ r1, y − 30 ∗ r2 > where r1 and r2, 0 ≤ r1, r2 ≤ 1, are random numbers. This random move is one of the cases that can be generated using the GSTD software [TN00] and is typical of moving objects. Assuming cars are moving at 100km/hour (= 30m/second), we choose 30m for the variation. The variation does not affect the update performance significantly because the update operation consists of independent two operations, delete and insert.”

– “The RUM-tree supporting frequent updates in R-trees using memos” [30]. This study does not consider transactional processing, but we include it to our survey because of its update inclination.

“We vary the percentage of updates from 0 (i.e., queries only) to 100% (i.e., updates only).”

The second one is the parameters to vary:

– Number of threads (sometimes called multiprogramming level) [28, 7, 3, 29, 2].
– Node size (also fan-out) [28, 2].
– The proportion of update and select transactions [28-30, 2].
– The probability of an individual operation for being a write one [7] or directly defining the contents of a transaction [29] (long and short transactions).
– Selectivity of a transaction [7, 2].
– Size of a transaction [7].
– Isolation level [29].

And the last aspect is what to measure:

– Throughput in transactions per second [7, 3]. Possibly, measure update throughput and read throughput separately [29].
– Throughput in operations per second [28, 2].
– Average transaction response time [3].
– Lock overhead (#locks/predicate checks) [7].
– Conflict ratio [7].
– Cache miss ratio [2].

Thus, we can see that various studies use quite different setups. Thus we are free to use any of the aforementioned approaches.

5.2 Our approach

Unfortunately, to the best of our knowledge there is no standard benchmark (like, for example TPC-C [31]) for transaction processing systems for multidimensional data.

Thus, we had to come with our own workload. We basically reuse the one offered by the organizers of the contest [32] with some modifications. The workload can be characterized as follows:

– Range transactions. This kind of transaction essentially reads 200 points of a requested rectangle.
– Insert, update and delete transactions. Each transaction consists of 5 operations of the respective type.
– Wildcard transactions are not considered.

Queries (as well as data) are pre-specified in a file and sequentially fed into the index. Range transactions are generated in the following way: we pick two points at random and use them to define the search rectangle. More detailed information may be found in the benchmark specifications. Also, we drop the sorted resultset requirement which was used in our previous paper [33].

Two phases were evaluated: index construction and query processing. During the index construction we measure data scale up with respect to a number of threads employed and the speed up.

Query processing phase was studied using the two following workloads:

<table>
<thead>
<tr>
<th>Update intensive</th>
<th>Select intensive</th>
</tr>
</thead>
<tbody>
<tr>
<td>range portion: 20</td>
<td>range portion: 85</td>
</tr>
<tr>
<td>update portion: 40</td>
<td>update portion: 5</td>
</tr>
<tr>
<td>insert portion: 20</td>
<td>insert portion: 5</td>
</tr>
<tr>
<td>delete portion: 20</td>
<td>delete portion: 5</td>
</tr>
</tbody>
</table>

We measured performance in operations per second.

5.3 Our prototype

Our system follows classical design guidelines and contains several high-level features:
An R-Tree data structure, which is built upon GiST [12], a popular template index structure which allows to “abstract” various tree data structures. It “abstracts” operations and applies concurrency control and transactional isolation mechanisms. This template allows to extend with the means of concurrent access almost any tree conforming to certain requirements. Some notable examples of the trees fitting into this template are $B^+$-Tree, RD-Tree and R-Tree [12]. “Abstracting” a tree structures via GiST is a convenient approach and is used, for example, in PostgreSQL.

Concurrency control and transactional isolation. We used a mechanism adapted from [11] with locks, latches and Node Sequence Numbers. Also we provided deadlock resolution mechanism. Eventually, we ensured the read committed isolation level. Details regarding our adaptation are presented in the next section.

In-memory index. Our prototype is designed for in-memory indexing.

Logging and Recovery. Currently our prototype lacks logging and recovery components.

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. This kind of processing (called logical deletion) is a widely-used approach for handling deletions in database systems [11].

We validated our implementation in two ways:

First, we used public third-party unit-tests, provided by [32]. We also had extended this test set with our own cases. These unit-tests ensured correctness of an isolation level (read committed) and other implementation issues.

An evaluation of our prototype with industrial systems PostgreSQL and Berkeley DB was presented in the reference [33]. This evaluation showed that the system’s performance is comparable to the industrial ones.

At last, our the first version of our implementation participated in the contest and was ranked 5-th on the public tests [32].

Our hardware and software setup was the following: hardware — 2 x Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz (32 cores total), 120GB RAM; software — Linux Ubuntu 3.13.0-30-generic x86_64, GCC 4.8.2.

We plan to release the source code to public soon. It would be put in the following url: ???

5.4 Experimental details

The workload parameters were the following: uniform data distribution, 4 dimensions, integer (8 bytes) datatype used. R-tree parameters were: plain R-Tree with 32 fan-out with Guttman Quadratic split algorithm [14].

We had employed two datasets:

Index construction phase (scale up) was studied using the a variety of workloads (128MB, 256MB, 512MB, 1GB index, 4 dimensions, uniform data distribution). We had to split scale up evaluation into two series of experiments
due to the memory and time usage. The 512MB index used almost 8GB of main memory and takes about a minute to construct (this goes in accordance with [2]). Unfortunately, we had to perform several runs in order to obtain errorbars, thus we had to limit the size of the index. The obtained data was averaged over 20 runs.

- Query processing phase (both workloads) and index construction (speed up) was studied using the 512MB index, 4 dimensions, uniform data distribution. The errorbars were calculated using the data from 10 runs.

6 Results and conclusions

In overall there were performed three types of experiments and $20 \times 7 + 10 \times 18 \times 3 = 680$ runs. We needed this amount in order to use statistical methods to perform repeatable evaluation and obtain credible results. We present the results in the paper [1].

7 Acknowledgements

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

References


25. Yi, K.: The priority r-tree. SIGSPATIAL Special 4(2) (July 2012) 8–12


