AALBORG UNIVERSITY
Department of Computer Science

TPR – tree
Running in Main Memory

Beata Nitkiewicz
Michał Pańczyk

August 2006

Abstract

As the branch of business, where locations based services matter, grows, the demand for reliable and fats spatio temporal databases capable of indexing dynamical data rises. The number of data structures able to handle this requirements is significantly small, thus the pressure on researchers grows to invent a simple and reliable solution. One of the idea to improve the performance is to move the traditional databases in to the main memory. As the memory chips prices are falling the idea is starting to become reality.

In this paper we investigate how the TPR-tree, as a data structure originally implemented to the disk oriented environment, behaves in the main-memory. We verify, wide-spread thesis of memory access time as the first-ordered bottleneck for in-memory indexes, by running experiments on the TPR-tree. Furthermore we stress that performance of the index structure running in main-memory is measured in different way and different indicators need to be considered. Experimental analysis leads us to the conclusion that the TPR-tree is to CPU heave structure and the simplest methods of improvements are achieved by limiting the number of needed computations. In order to demonstrate the path for the future analyses we introduce three different improvements for the update algorithms: *lazy delete*, *bottom-up delete* and *new penalty insert*.

Preface

The project was writen during the Spring Semenster of 2006.

We would like to thank our supervisor, Simonas Saltenis, for his invaluable support, guidance and valuable comments during the project work.

24. August 2006

Beata Nitkiewicz

'---
Michał Pańczyk

Contents

1 Introduction

With the recent advances in mobile computing devices (e.g. cellular phones, PDA, and GPS) as well as wireless technology and positioning systems, the importance of moving object environments have grown significantly. Numerous of real-life applications, providing location based services (LBS), such as mobile communication management, traffic monitoring, intelligent navigation, integrated information services, e.g., tourist information services, and location-based games require enormous number of moving objects to be managed and queried. The main problem of these applications, employing disk as mass storage, is rapidly track the positions of moving objects and effectively supporting queries. As the hard drives bandwidths are relatively small for the numerous operations of spatio-temporal databases a different solution has to be considered.

In a science-fiction story from 1986 “Johnny Mnemonic”, William Gibson uses 40 megabytes as shocking, unreachable size of memory. In 1995, in the film adaptation this value has been changed to 40 gigabytes to still shock the audience. Nowadays the size does not seem any fantastic and certainly would not be used in any of science-fiction novels. While the prices of memory chips are dropping and one can buy more and more megabytes of memory for the same price, in the near future megabytes will have to be changed in to gigabytes and then terabytes etc. Thus, it is becomes reasonable to have databases system with large amount memory and the idea of main-memory databases is not science-fiction any more. In such environment, the entire database or most often used parts can be moved in to the main-memory.

In previous years many researches were conducted to optimize the disk oriented data structures. However a new approach, to put data structures in to main memory, started to come into prominence. Ailamaki et al. [3] show that for database resident in memory, half of the execution time is spent on memory access. Since then, the studies on main-memory indexes focus mainly on making structures cache concious and minimize L2 cache misses [2, 22, 9]. In order to that the node size used in these indices is defined to be equal to the L2 cache line. As Hankins and Patel [10] shown this design ignores the large number of instructions executed on data elements after they are read into the CPU cache. This may be very important, especially in such complex spatio-temporal indexes as the TPR-tree.

In this paper we present the results of researches on Time Parametrized R-tree

(TPR-tree), proposed by Saltenis et al. [14], working in main memory environment. To the best of our knowledge this is the first such an attempt made for the TPR-tree. We choose the TPR-tree as the effective data structures for indexing moving objects in multi dimensional space. TPR-tree does not involve any space transformation and indexes data in intuitive way using the time-parametrized lineal function as movement representation. Modelling the position of moving objects as a function of time allows to answer future queries and reduce the frequency of updates.

In our researches we investigate the influence of changing the environment on the performance of update and query operations. We follow the researches of Ailamaki et al. in [3] and prove that for such a complex data structure as the TPR-tree memory access is not the bottleneck and takes less then 10% of total time independently of node sizes. Since the entries, basic components of nodes, keep the so-called time-parametrized bounding rectangle, the node size is always larger than the cache line size, even for a smallest node capacity. We investigate the evolution of existing index structure and point that these structures and their algorithms are getting more and more CPU heavy to archive better disk I/O performance. We show that once the tree is in main-memory resident, significant improvements can be made by simplifying the TPR-tree algorithms with only slight decease of the search performance. We introduce three new techniques; two for delete and one for insert. They crux is to reduce the number of required calculations. The improvement is significant as well for insertion as for deletion.

The reminder of this parer is organized as follows. In Section 2 we address the issue of moving objects indexes and present TPR-tree genesis and structure. Section 3 discuses the evolution of databases from disc oriented to the in-memory databases and the main issues involving the new environment. In Section 4, we overview the main memory indexes. Section 5 provides our idea of improvements. The experimental results are discussed in Section 6. While the conclusion is presented in Section 7, future work is discussed in Section 8.

2 Indexing of Moving Points

In this section we first describe the reasons why the issue of indexing moving objects is so important nowadays. Then we outline the data that need to be indexed and distinguish three types of queries. Next we specialities of movement representation. Finally we describe the R-tree[19], R*-tree[18] and TPR-tree[14].

2.1 Why do we need Moving Objects Indexes?

In the case of applications dealing with mobility, the main task is to keep track of where air planes, boats, cars, vessels at sea, people, and many other assorted moving objects are at any point in time. The consequence of mobility is continuous change of location thus keeping data accurate to the reality means that moving object's location has to be continuously updated. The main assumption of traditional database management systems (DBMSs) is that data stored in database remain constant unless it is explicitly modified. Thus, the model where updates are issued in discrete steps, at every unit of time is obviously inefficient and unworkable solution. If DBMS were to update continuously changing location, it would imply a discouragingly high update overhead. This is the reason why the moving objects indexing techniques are becoming more and more important. To avoid all the problems mentioned above, Wolfson et. al. [13] propose an approach to model each object's location attribute as a function of time $\vec{x}(t)$ and update the database only when the parameters of the function change.

2.2 Movement Representation and Query Types

Modelling the position of moving objects as a function of time means that we have data representing movement of the objects. The most popular representation is by linear function, because of two non-trivial reasons:

(1) minimum number of parameters required,

(2) possible description of more complex movement by using interpolation [12].

Since the database stores the information about the moving objects, and objects update their positions, their significance in the system is essentially equal to the others' components. They become the part of the system and are responsible for updating the database whenever the parameters of their movement change. Information are transmitted via wires networks. Because the required number of parameters is minimized we can limit the necessary wireless bandwidth between the database and moving objects. That has great impact on the communication cost, which is the component of total update cost[17]. Thanks to the (2), this relatively simple data representation does not limit us to one, and only one type of movement, where roads are straight and speed is constant. By interpolation of the gathered data we can manage more complex types of movements.

Representation of movement by linear function requires only two parameters, which are data that need to be present in the index structure:

- object's initial position,
- velocity vector.

In general, an object can be moving in d-dimensional space using some complex motion, but in real world objects move in up to 3-dimensional space. However, without loss the generality, in this paper we focus our attention on 2-dimensional spaces where the motion of objects is represented by linear function.

The first parameter of the function $\vec{x}(t)$ is an object's position at particular time t_{ref} and it is denoted as $\vec{x}(t_{ref})$. The time t_{ref} is a reference time at which the object is stored. The second parameter - velocity vector is denoted as $\vec{v}(t_{ref})$. Thus, only the tuple $(\vec{x}(t_{ref}), \vec{v}(t_{ref}))$ needs to be stored in the database and the movement of any objects is represented by Equation 1.

$$\vec{x} = \vec{x}(t_{ref}) + \vec{v} \cdot (t - t_{ref}) \tag{1}$$

When an update of an object arrives at time t_{up}, the new position for the object is calculated using the Formula 1, in which t_{up} becomes new t_{ref}. Position and velocity values are easy to obtain, directly from the GPS[16].

There are two very important advantages of that solution:

(1) the reduced amount of updates,

(2) the possibility of computing the location of an object in the near-future.

The first advantage has been proved by Civilis et al. [17] by comparing different update policies for moving vehicles, among others, so called, a point policy and a vector policy. In simple point policy positions of an objects are stored as a points and the current position is given by the most recent update. Vector policy deals with linear function as a movement representation and uses pair of direction and speed for a position approximation. The conclusion results from comparison indicate that prediction of the future position allows to significantly reduce the amount of updates. Authors have shown that for the accuracy threshold below 200 meters vector policy is more then two times better then point policy.

The second advantage is very useful considering the predictive queries, such as "Give me the list of all air planes over London in the next half an hour". By modelling the moving points with a function of time we can predict the future position and this way the answer for the query above is more accurate to the reality.

It is worth mentioning that the pair of parameters can be used not only for stored objects. In the TPR-tree [14] the velocity vector and reference position are also used for representing the coordinates of bounding rectangles as functions of time. We will discus that, with more details in the subsection dedicated to TPR-tree.

Moving objects indexes should support queries retrieve all points within specific region. Following the distinction in [14] we identify following types of queries:

- Timeslice queries return all objects that intersect a given rectangle at specified time t_1, after the current time CT, where $t_1 > CT$.
- Window queries return all objects that intersect a given rectangle at same identified interval time from t_1 to t_2, where $t_1 < t_2$, where $t_1, t_2 > CT$.
- Moving queries return all objects intersect a given moving rectangle sometime between t_1 and t_2, where $t_1, t_2 > CT$.

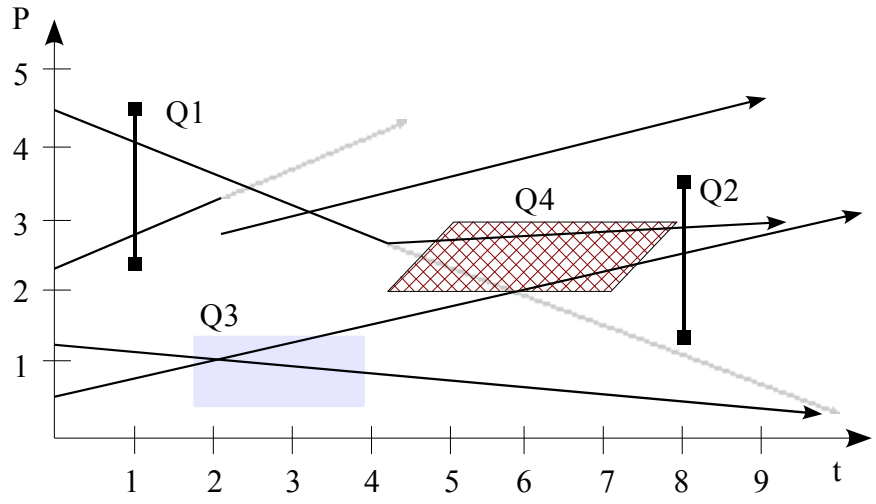


Figure 1: Query Examples in One Dimension Space

In Figure 1 the examples of three query types, in one dimension space, are presented. While the x-axis symbolises the time, the y-axis represents the one dimension object. O1 and Q2 are time slice queries, Q3 – window query and Q4 – moving query.

2.3 The TPR-tree genesis

Although the R-tree and R*-tree not prepared to index moving objects, they are the basic spatial indexes and the foundation for TPR-tree. We discuss the R*-tree and R-tree due to their influence on TPR-tree structure. Reader interested in an extensive survey of R-tree evolution, its variants and implementation issues is referred to the paper [21].

2.3.1 R-tree

The R-tree (Region Tree) was proposed by Guttman in 1984 [19]. Since then, together with its descendants it has become one of the most widespread access methods. R-tree applications range from spatial and temporal to image and video (multimedia) databases [21]. The R-tree is an indexing method for multidimensional data, such as point, line segments, surfaces and volumes in high-dimensional spaces [19]. Data, that are d-dimensional geometric objects, are represented by minimum bounding d–dimensional rectangles (MBRs). Applying this approach, R-tree stores data without clipping them or transforming them to the higher dimensional points. Each internal node corresponds to the MBR that bounds all its children and contains entries of a form (R,p), where R is the MBR that contains the MBRs belonging to the child of which, a node is pointed by a pointer p. Leaf nodes contain entries of a form (R,o), such that o refers to the database object, and R is a MBR that contains that object. The root of a tree has no less then two children (unless it is a leaf) and every other node has between m (minimum number of entries) and M (maximum number of entries) entries, where $1 < m \leq M/2$. Figure 2 depicts some objects and a spatial query (shaded rectangle) on the left and the corresponding R-tree on the right. 10 data rectangles *D* through *M* are stored in leaf nodes, whereas MBRs *Ra*, *Rb* and *Rc* are in the internal node.

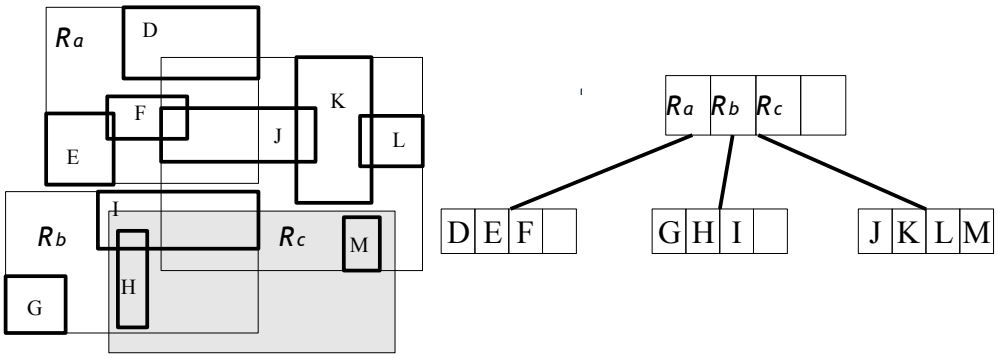


Figure 2: R-Tree.

The MBRs that bound different nodes might be overlapping, as it is shown in the Figure 2: *Rc* partly overlaps *Ra* and *Rb*. Furthermore, there can exist containment relationship between rectangles. This means that search algorithm, which descends the tree from the root, may need to search more than one subtree under the node visited. For example, in the Figure 2 the shaded rectangle defines the spatial query, which should return the objects that even partly overlap the query area. On the level $R_a/R_b/R_c$ the two MBRs are chosen - R_b and R_c . Then the searching algorithm traverses two subtrees rooted respectively in R_b and R_c. As the answer for the query three objects are returned: H, I, and M.

The R-tree is dynamic (insert and delete operations can be intermixed with select operations) and balanced (all leaves appear on the same level). Since it is dynamic a non-periodic reorganisation becomes unnecessary.

The R-tree insertion algorithm can by divided into two steps. The first one is to choose insertion path and the second one is to split overflowing node. The first step uses heuristic optimization method to allocate new entry and to find optimal MBR for it. The MBR that required the smallest area enlargement is chosen. The second one uses the optimizing heuristic by minimizing the area sum of the two rectangles resulted from the split. Guttman presents three alternative split algorithms of linear, quadratic and exponential complexity with respect to the number of entries of a node. Essentially, all the variants of R-tree that have been proposed later differ from the original one in the way they execute the split by proposing different minimization heuristics.

2.3.2 R*-tree

The R*-tree have been proposed in 1990 [18]. Since it has become a well-known access method widely accepted in the literature and commonly used as a fundamental assumption for performance comparisons [14,23]. Basically, the main differences between the R*-tree and the R-tree are the optimization criteria. While the R-tree minimizes the sum of the area of the produced MBRs, the R*-tree takes under consideration additional criteria, such as:

- minimization of the overlapping between MBRs at he same level,
- minimization of the margins of the MBRs,
- optimization of the storage utilization.

Other novel feature of R*-tree is a new technique, so called 'forced reinsertion', which is used during the insertion routine. When a node overflows and it happens first time on a certain level, R*-tree forces p^* entries to be reinserted instead of directly splitting the node. If any of p entries could be assigned to other node a split is avoided. That has an effect on the split occurrence frequency as well as on the overlapping between neighbouring nodes. Split takes place quite rarely and the overlap decreases. Thus, the heuristic approach applied in the R*-tree improves insert and split methods. That leads to a better tree structure (in comparison to the original R-tree) in the sense of the retrieval performance.

The delete algorithm of the R*-tree first identifies the node that contains the entry to be removed and then two situations are possible. (1) There is still more then minimum number of entries in the node. Deletion terminates. (2) Node generates an underflow. In this situation the node is deleted and all the entries of the node are reinserted by regular insertion algorithm. Underflows may propagate to upper level and are handled in the same way.

2.3.3 The TPR-tree Structure and Algorithms

The time-parametrized R-tree (the TPR-tree, for short) [14] is an extension of R*-tree, which efficiently supports querying moving objects in current and estimated future time. The structure of the TPR-tree is inherited from the R*-tree [18]. Every entry which is not in a leaf node contains the pointer to the child node and MBR which tightly encloses the entries in the child node in a current time. The Figure 3(a) presents the MBRs in $t=0$, where $t >$ current time.

For indexing of moving objects TPR-tree employs linear function as a

* p being the parameter. Optimal value for that parameter have been obtained experimentally by Guttman et al. [18] and that is equal to 30% of M (maximum capacity of the node).

representation of objects' movement (See Section 3.2). It means that object's position and velocity vector are stored. Based on these two data one can compute current and projected future position of an object.

Moreover the decision about the assignment of objects to the bounding rectangles is made on the basic of the objects' position and their velocities. The Figure 3(a) shows the representation of the MBR R_1 and R_2 each contains two objects, which are MBRs too. The arrows denote the directions of their edges velocities, while the numbers correspond to their values. Negative values imply that the velocity is toward negative direction of the axis.

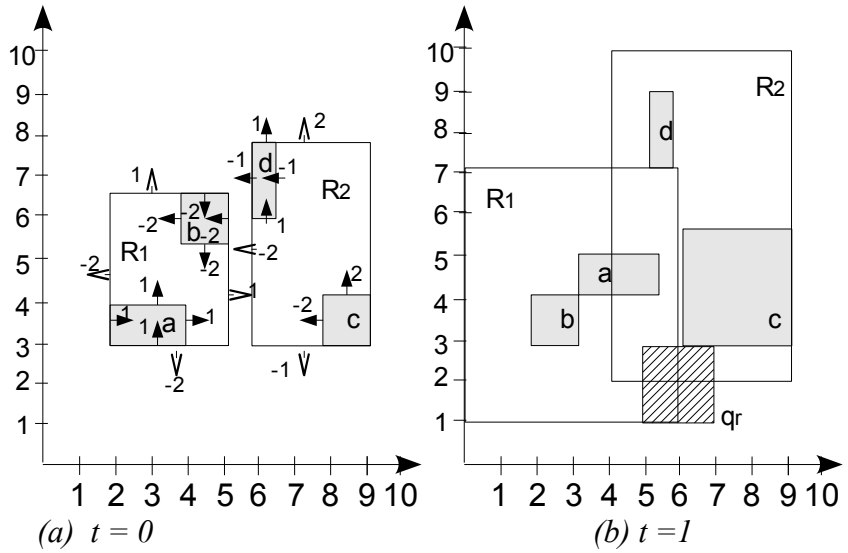


Figure 3: TPR-tree

One of the basic conception of the TPR-tree is to employ time-parametrized MBRs. Since the coordinates of the MBRs are function of time, the bounding rectangles follow moving point or other rectangles as they move. That indicates that they are moveable, they have their velocity vectors which are determined in the way ensuring that the MBR always enclose the underlying objects. Specifically, upper (lower) bound of MBR is set to move with the maximum (minimum) speed of all speeds of enclosed objects on this dimension. Figure 3(b) shown a, b, c, d and enclosing them nodes R_1 and R_2 at time 1. Their positions and sizes are different then at time 0 (Figure 3(a)). The MBRa are not minimum all the time but only at the moment when they are created or updated. TPR-tree uses, so-called conservative bounding rectangles, which are minimum at certain time, but never shrink and almost always grow larger over time. In the case when all the enclosed point move exactly in the same way, conservative bounding rectangle move with them and their size remain unchanged. As a time passes the sizes of MBR becomes very large and are generally larger then strictly needed. That can drive to high region overlaps which

greatly decrease the query performance. To forestall that it would be desirable to correct MBRs ones in a while. The TPR-tree tightens them up every time when any of the moving point or rectangles is updated.

The TPR-tree supports all the queries presented in Section 3.2. In order to answer the queries about location at future time t, it employs linear function (see Section 3.2) to calculate predicted object position as well as MBRs' future extends.

Considering indexes that predict future position the issue of how far in the future queries can ask needs to be taken under account. The TPR-tree employs for that propose parameter called querying window (W) as well as two other parameters. (1) Index usage time (U), which is a time interval during which an index will be used for querying, and (2) time horizon (H) which is the length of the time interval from which all the time instances specified in the query are drawn. Time horizon H is equal to the sum of index usage time (U) and the querying window (W).

Let t_l denote the time when the index is created or loaded. During the interval time [t_l , t_l + H] all MBRs should be as small as possible. The insertion algorithm of TPR-tree is an extension of the R*-tree algorithm, which has been adapted to deal with moving points. The aim was to minimize three objective function, such as:

- areas of the bounding rectangles,
- their margins,
- overlap among the bounding rectangles.

These functions are time dependent (let A(s) denote such a function), and TPR-tree considers their evolution in [t_l , t_l + H] . Therefore, the main adoption was the minimization of the following definite integral instead of minimization of the objective function.

$$\int_{tl}^{tl+H} A(t)\,dt \quad (2)$$

Thus, the TPR-tree insertion algorithm is the same as the R*-tree's, but instead of above functions the integrals of those functions, as in Formula 2, are computed. The algorithms for that are presented in [28]. It is worth to notice that split algorithm uses by TPR-tree takes under consideration velocity vector to make the bounding rectangles grow more slowly.

The TPR-tree delete operation is performed exactly in the same way like in R*-tree (see Section above). When a node becomes under full it is deleted and its entries are reinserted.

3 The Idea of Main Memory vs Secondary Storage Databases

In a case of highly critical database monitoring the trajectories of planes there is no place for results of queries arriving too late and putting human life at risk. Consideration all the factors influencing response time of that database became essential. In this section we show those factors and explain the evolution of the high load databases, its bottlenecks and commonly used solutions. First we present databases working on disk structures and the reasons why evolving to databases located in main memory (main memory database system – MMDBMS) is a good solution. Then we point the main bottleneck of MMDBS which is the memory access as presented by Ailamaki et. al. in [3]. Because the cache memory takes part in overcoming that problem we briefly describe the idea of CPU cache. Then we present how the cache works and what is the most important part of the cache from the databases' point of view. Although we need to remember that all the hardware solutions without proper utilization are not going to work by itself.

3.1 The Typical databases – Disk Oriented

Typically, when we think of databases we mean secondary storage databases, because nowadays it is the most commonly used type of databases. In that kind of databases the data are stored on external mass memory, i.e. a hard drive. This way there is no limitation of capacity - one can have a single hard drive or a farm of servers seen as one logical disk. By overcoming that limit we can keep any kinds of data that we can imagine – i.e. a university database managing the employees can keep not only the names, addresses and their positions, but also the photos, videos from conferences etc.

Problems may occur when we would like to use disk structures for high load databases. The performance ratio will drop down. Each time that an operation is preformed there has to be some input or output (I/O) operation. As Rosenblum, et al. points in [101] this operations are the first-ordered bottleneck for that kind of databases. It is because of the stalls – when the system has to wait for the data to arrive from the hard drive. That involves a mechanical movement of the head of a hard drive and pointing it to the right position. That time is measured in milliseconds but multiplied by a number of operations grows to a significant value [26].

In the 1990s the percentage of time wasted on I/O operations has grown from 10% to almost 90%. That was the impulse for the researchers to start optimizing the disk structures in order to decrease the I/O latency. When that was not enough many efforts were spend moving the databases into the random access memory.

3.2 The Idea of Main Memory Databases

The times where 640 kB were enough for everybody are gone. Nowadays we observe a trend that one dollar buy us more and more memory each year [21], so the memory capacity is not the limit any more. That is why we can easily think of putting a whole database into RAM. This way we get rid of all the disk databases problems that we have mentioned in the section above. With that the latencies are significantly smaller – counted in processor cycles.

So why is the whole world still using the standard databases ? There are couple of reasons for that and we will just point the most important:

- Price – one dollar buys more capacity of disk than of main memory. Not everybody is willing to pay the price for the extra speed. For most purposes the standard computer systems are enough,
- RAM is volatile memory. It is not considered reliable because when power supply or system stability is shaken our data are at great risk of being lost forever. Uninterruptible power supply (UPS) is a simple fix for the power problems [20]. There are not really many solutions for the system instability except for the backup/transaction logging systems which hurt the performance,
- Capacity limits. Thanks to 64 bit architecture operating systems can address up to 2^{64} bytes (16 exabytes) of memory. But the real life limits are far lower than that – there is no hardware supporting that large amounts of memory. In the environment where capacity is limited the set of the data to be stored need to be carefully selected. One of the solutions is to split the data into, so called *'hot'* and *'cold'* parts. Hot data are accessed frequently and usually low volume, while cold ones are access rarely and more voluminous [29]. Considering the system tracing the buses in the city, the example of hot data is the current position and cold data is a type of the bus (brand, capacity).

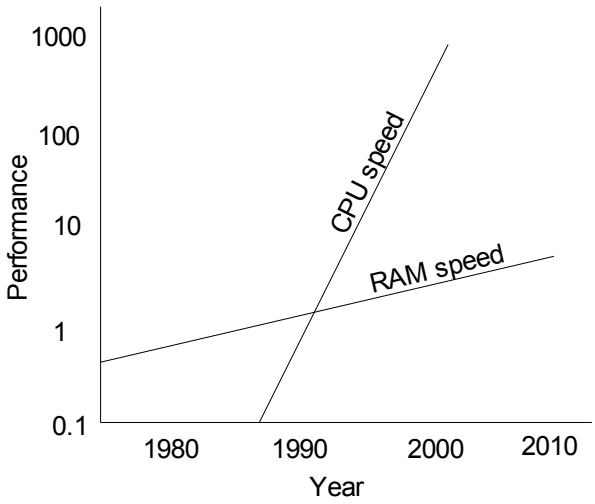


Figure 4: Modern bottlenecks for in-memory databases. Based on [25]

Moving the database from disk to main memory reduces the main bottleneck. We switch the store from the hard drive to RAM. This way we eliminate the key factor – the I/O operations to the hard drive.

But still, overcoming one bottleneck leads us to another one. As shown in the Figure 4, since the 1990s a gap between the performance of the CPU and the memory has been observed. As McCalpin [25] predicts that gap is growing and it is becoming to be the main bottleneck of the current programs operating in main memory. So the goal is to limit the number of accesses to the RAM. We are going to explain why and how much is it going to cost us in the next paragraph about the cache memory.

3.3 The idea of CPU cache

When we look deeply in to a Central Processing Unit (CPU) we can see an equivalent of a computer system (see Figure 5). It has a processor - algorithmic and logical unit (ALU), own memory – registers, and communicates with external storage (i.e. RAM memory) using I/O lines.

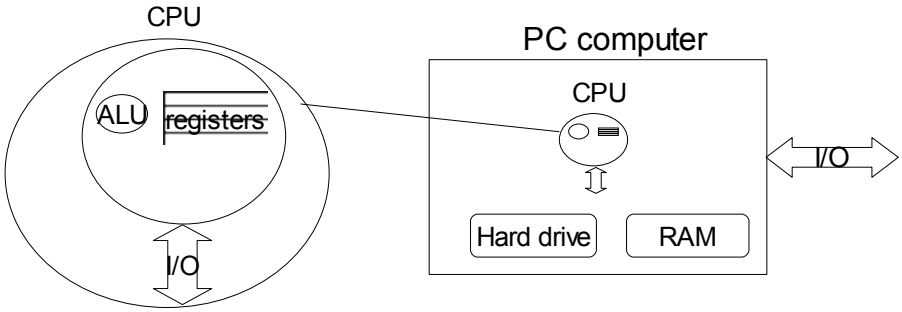


Figure 5: CPU as a computer system

ALU does pure calculations. That's the fastest part of the computer - its basic operation time is a processors tick (1/Processors Clock speed in Hz).Every operation requires to load the executed data from the RAM – memory a lot slower than the CPU. Every time that the data has to be read in from the RAM the processors time is wasted. To reduce the number of that kind of situations a go-between was added – the cache. It is a very fast (comparable to the CPU speed) but small memory build in to the processor that buffers data flow between RAM and the processor. In modern computer systems cache is split in to at least two, sometimes three parts, more commonly called levels, as shown on Figure 6.

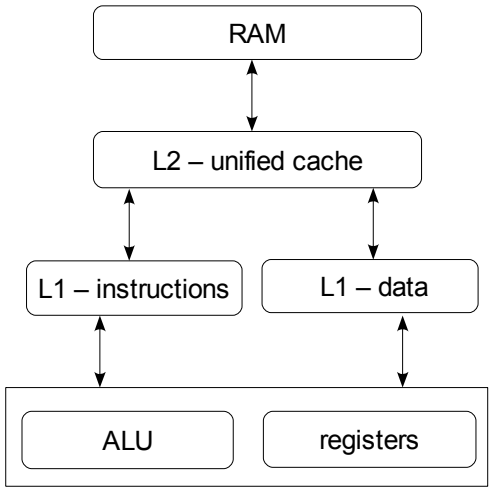


Figure 6: In-memory communication

The Level 1 (L1) is the one nearest to the registers and it is the fastest .It's size varies from 32 to 64 kB. It is split between instructions and data. The second level (L2) of cache is unified (instructions and data are kept together). Its size varies from 256 kB to 4 MB. Usually this part has the access to the RAM but some times level 3 is added.

3.4 The Way the Cache Works

We are not going do go deeply in to cache operation problems and solutions but we just want to show the parts that we had in mind during the project. We omit the table look-aside buffer (TLB) on purpose because we have no control over that in our project. The following section has been based on [24] and on [28].

The cache works as a buffer. Each time that the next instruction of the code is executed it has to be read from the program's memory and written to a register. Without the cache, once the address was bound, the CPU would have to wait for fetching the data. Once the instruction is loaded into the register the processing starts.

Because programs are a code put into the memory it is highly probable that the instructions lie one after another. So small but very fast part of CPU called branch prefetcher analyses the code, makes predictions of which part is going to be needed and reads a set of instructions in to the cache. More specifically it reads the set in to L2 and takes the currently needed part closer to the registers, to the L1. So now processing of our code is speeded up - once we have the address we call for the data that it carries. First it is checked if it is contained in the L1 cache. If not we call to check if L2 contains it. If not we still need to call the RAM - but now we do not grab a single instruction but block of instructions – highly probable candidates for the next parts of the code.

All the calls to cache and RAM have a cost measured in processor cycles. Finding an instruction in the cache is called a cache-hit. A cache hit on in the L1 takes one cycle. If the instruction is not in the L1 that is counted as a cache-miss and the L2 is queried. Finding an instruction in L2 and loading takes, depending on the architecture and operating system, from 10 to 20 cycles. A cache-miss in L2 costs from 100 to 200 cycles. So the speed up in a optimistic situation is around two hundred times.

3.5 The Most Important Part of Cache for in-memory Databases

In a heavy load databases where we operate on huge amount of data it is impossible to have everything cached. The instructions to data ratio drops down. Every part of information included in the DB has to go through the CPU – from RAM via cache to the CPU and go back in reverse order. The write to the memory is

buffered in the cache so it is handled once in a while (when the buffer is full). But when we need to execute some parts of the code the data is needed right away. Each time it is not cached, a call to the RAM needs to be made – that means that the CPU is unused. We cannot expect the huge amount of data, used in spatio-temporal database, will be present in the L1. The L1 cache misses penalty is necessary cost we have to bear. But we cannot afford continuous cache misses on the next level of cache. We have to minimized the number of those misses. This is possible – especially when we use good operating system, the program is written according to cache-conscious programming rules [28], and we used a good compiler.

While we do not have great impact on the behaviour of L1, we can program the code, so that the utilization of L2 is going to be optimal. This optimization is especially important in the light of researches presented in reference [3]. Ailamaki et al. have shown that in the database management systems in main memory the main part of execution time is spend on second level data cache misses and first level instruction cache misses. That inspired us to to verify their conclusion for complex multidimensional data structures in main memory, such as TPR-tree.

4 Existing Indexing Techniques for Main Memory

In this section we first explain two main techniques used by existing cache conscious trees. We next categorize indexes according to the compression technique and present most widely known cache conscious trees, namely, the CSB^+-tree [2], the pkB^+-tree [4], and the CR–tree [5]. We do not give details about the algorithms, (such as insert, delete, and search) used by the indexes. The interested reader is referred to the papers written by the indexes' authors [2, 4, 5]. Instead, we describe the structure and the most important principles of the compression used by given index.

Cache conscious indexing techniques can be distinguished into two different categories - those based on pointer compression (also so-called pointer elimination) and those based on key compression [1]. The idea of cache conscious trees is to increase the blocking factors of the node in the indexes. The most important factor influencing the performance of main memory DBMS is cache miss.

4.1 Pointer Elimination Technique

The key idea of cache conscious trees grouped into pointer elimination category is to eliminate most of the child pointers from the node to increase the blocking factor in the internal nodes. The pointer compression effectively reduces the tree height, and thus improves the cache behaviour, when the node size is set to cache line size – the natural transfer size for reading and writing the main memory. The examples are Cache Sensitive B^+ - tree (CSB^+ - tree) [2]. The efficiency of pointer elimination depends on the relation between key size and pointer size. If the size of the key is much larger then pointer size the effect of the pointer compression technique is poor.

Cache Sensitive B^+ - tree. In 2000, Jun Rao and Kenneth A. Ross proposed a structure called CSB^+ - tree. It is a variant of B^+ - tree [8]. Figure 7 shows an example of CSB^+ - tree. Like B^+ - tree, data is stored in the leaf nodes. The key difference between CSB^+ - tree and B^+ - tree is the structure of internal nodes. There is only one pointer in a non-leaf node. This pointer references the *node group*, which is represented by a dashed rectangle in the figure. Each node in a group is stored physically consecutively. Since the child nodes are allocated in that way in main memory, the parent node stores only pointer to the first node in a node group, which is represented by the solid arrow in the figure. The address of a child nodes can by computed from that pointer. By eliminating pointers to the child nodes in internal nodes, there is more room for additional keys and hence better cache performance.

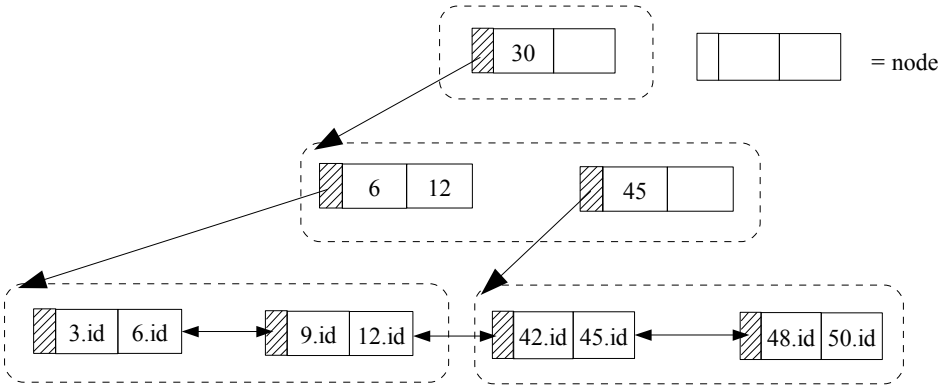


Figure 7: CSB+ - tree.

4.2 Node Compression Technique

In multidimensional index structures such as the R-tree[19] typically size of key, an MBR (minimum bounding rectangle), is much more larger then that of the pointer thus simple four-bytes pointers elimination technique does not help much to increase the blocking factors in the internal nodes and leaf nodes. Compression of MBR keys, which occupy almost 80% of index data in the two-dimensional case[5], reduce the space for entries and that way it is possible to pack more entries in a node.

Multi-dimensional index structures can be grouped into two approaches according to their space reduction method [6]. First approach is to compress minimum bounding regions (MBRs) by quantizing coordinate values to the fixed number of bits. Second approach is to represent MBRs relatively to its parent MBR. The examples are pkB-tree [4] and Cache-conscious R-tree (CR-tree)[5].

PkB-Tree. The pkB-tree is a modification of the B-tree that uses a key compression. It performs that compression by managing only the different part between the base key and the key to be compressed. Its structure is identical to B-tree except for the structure of index keys. Each key is represented by tree items:

- a pointer to the data record containing the key,
- the offset of the first bit at which the index key differs from base key,
- the first*l* bits of index key following this offset [4].

Internal nodes contain index keys and pointers to the subtree, while leaf nodes contain only index keys. Figure 8 shows an example of pkB-tree, where dashed arrows denote the base keys for index keys and solid arrows represent pointers to child nodes.

Bohannon et. al. [4] have proved that partial-key trees incur fewer cache misses than B-Trees with all beside the smallest key sizes. The above statement is true, but pkB-tree is limited only to low-dimensional data.

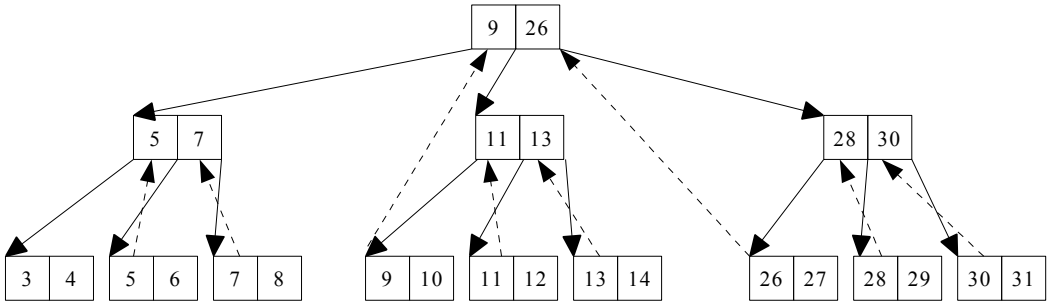
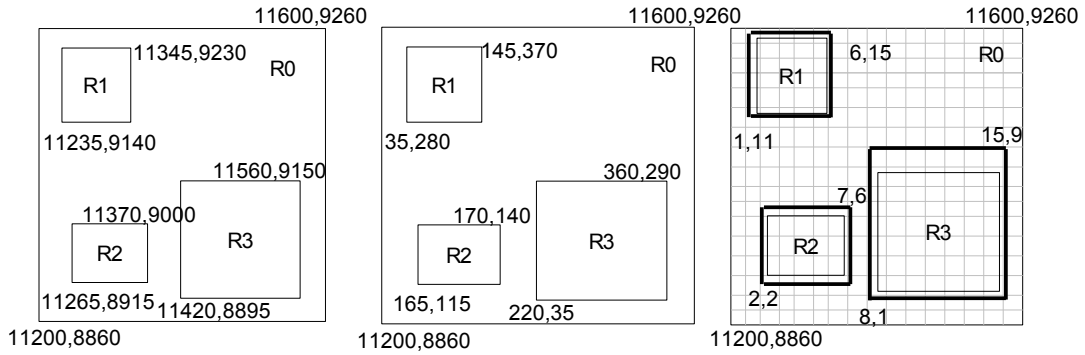


Figure 8: pkB -tree.

Cache-concious R-tree. The basic idea of CR-tree is to widen index tree by compressing MBRs so as to make R-tree[19] cache conscious. The CR-tree makes use of the compression schema called Quantized Relative Representation of MBR (QRMBR). Since the coordinates of the QRMBR have smaller size then those of the actual MBR, the MBR compression technique of CR-tree increases the fanout of nodes.

The key compression technique used by CR-tree first represents the coordinates of any given MBR key relatively to the lower left corner of its parent MBR and then, it quantizes the relative coordinates with a fixed number of bits.



(a)Absolute coordinates (b) Relative coordinates (c) Quantized relative coordinates Figure 9: QRMBR Technique

Figure 9 shows an example of QRMBR in the CR-tree: Figure 9(a) shows an absolute coordinates of the out most rectangle R0, which is the parent node's MBR, and the inner rectangles R1, R2, and R3, which are child nodes' MBRs. Figure 9(b) shows the coordinates of R1~R3 represented relatively to the lower left corner of R0. Figure 9(c) shows the coordinate values of R1, R2, and R3 quantized in 16 units, which require only 4 bits. The QRMBR enlarges slightly MBRs to align MBR to quantized coordinate values, which is represented in the figure by the thick rectangles surrounding the child nodes' MBRs. The space to store the coordinates have been reduced thus the maximum number of entries become larger then normal R-tree. QRMBR technique is a lossy compression scheme, which means that compression algorithm actually reduces the amount of information in the data, rather than just the number of bits used to represent that information. Lossy compression can cause the anomalies, i.e., it can choose a wrong insertion path [9] or search performance can be poorer than the original R-tree. The second anomaly can be avoided if the QRMBR schema is applied only to the internal nodes and the leaf nodes store actual MBRs [5]. This extension of CR-tree is called FF (false-hit free) CR-tree. In the paper defining CR-tree[5], the authors propose also two other variants of CR-tree: PE (pointer-eliminated) CR-tree and SE (space-efficient) CR-tree.

The PE CR-tree employs pointer compression technique, because the size of the key in CR-tree is now small unlike in R-tree. That increases the fanout of the internal and leaf nodes, but on the other hand, like in CSB^+-tree, node split becomes much more expensive. Split operation creates new node, that node has to be kept consecutively with its sibling, thus needs allocating and deallocating memory.

The main idea of second variant of CR-tree is to eliminate the reference MBR from nodes of the PE CR-tree. The elimination have been used because reference MBR of a given node can be computed from the matching entry in its parent node.

5 Our Idea Based on Previous Reaserches

In this section we show our ideas of solving the performance limitations of the TPR-tree, based on previous researches. First we present the idea of adaptation of the TPR-tree to the main memory environment. Next we discuss the impact of the node size for the performance of the in-memory index structures. We notice the complexity trends and heuristics, and to finally discuss our ideas of improvements.

5.1 TPR-tree in Main Memory

Not many of up to date moving object access methods are directly dedicated for the main memory – we discuss some of them in the Section 4. Mostly spatio-temporal indexes were implemented and tested in so much different disk environment where the number of I/O operations was considered as the main performance factor. Most of researches have been focused on reducing this significant bottleneck.

When speaking about in-memory indexes two aspects need to be distinguished: the theoretical and the practical one. The first one consists of rules about the tree structure and algorithms design. Usually cache concious index structure is created form an existing index. It is adapted to the main memory by modifying the structure with a pointer or key elimination techniques (see Section 4). The algorithms are then corrected to suite the new structure. The practical part is the way that the index is implemented and depends on several factors, such as programming language and style, the operation system, the platform etc. The details of our implementation are described in experimental section.

In this paper we concentrate on TPR-tree. Considering the theoretical rules about the tree structure ans the algorithms, there are no barriers for implementing it in the main memory environment. While Saltenis et al. in [14] have focused and optimized TPR-tree for the disk, our researches concern main memory implementation.

In order to reimplement any disk resident index structure to the main memory one has to take under account the different characteristics of those environments, such as the capacity and average access times. Looking at the memory hierarchy (see

Figure 10) the memory speed is a trade off of capacity. Hard drives have a basic unit of memory equal to a page – which is usual 4 Kb or more and the latency times measured in milliseconds. The situation with cache is completely opposite. The memory is ultra fast (measured in nanoseconds), but the capacity drops down – the basic unit is a cache line – depending on the processor from 32 to 128 bytes.

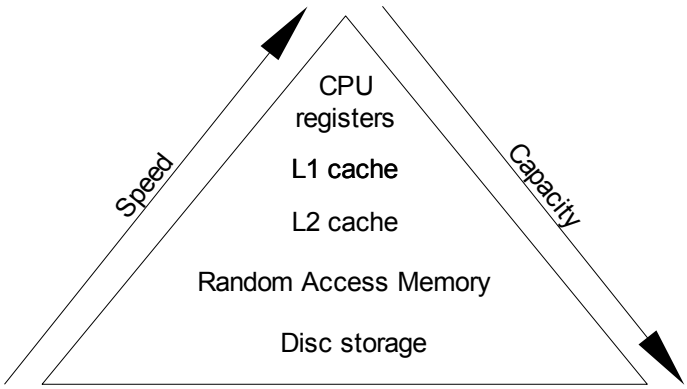


Figure 10: Memory hierarchy based on [23]

As mentioned in the Section 3 the RAM latency can be decreased or almost completely eliminated when the cache is properly utilized. So the main solution is the optimal cooperation with the cache, which guarantees that currently needed data are included in the cache. If they are not - the programs suffer the consequences of the average memory access time growth. While the natural data transfer unit in the main memory is the cache line, the basic of a tree structure is a node. Consequently, the node has to be optimized to fit in one or a multiplicity of a size of the cache line [27, 28]. Additionally, the node should be a compact structure so that it utilizes all the occupied lines.

It is worth to notice that implementation details are also important. It appears that, some of them, for a disk base index structures are completely trivial and come into prominence in main memory. We discuss our implementation deeply in Section 6.

5.2 Node Size Compared to the Cache Line

What we found in many previous researches was a correlation between the size of the node and the performance of tree indexes. As Rao and Ross showed in [27] the performance was almost optimal when the node was fitting the cache line. Their researches were done on a B^{+-}tree which carries far less information then

multidimensional data structure like TPR-tree. In case of TPR-tree a single entry is large in term of bytes (in our implementation it is 44 bytes). Thus a node as a set of entries does not fit to cache line. The same authors in [2], have proposed CSB^+ - tree (see Section 2.1). They run experiments in the environment where the cache line was 32 bytes. One cache line was enough to keep a full node containing 14 key, a child pointer and the number of keys used. It is worth to notice that even a single TPR-tree entry, occupies 44 bytes*, certainly would not fit the size of a cache line.

In opposition to Rao's research, Hankins et. al. [10] had different experiences with CSB^+tree. Their experiments imply that nodes of sizes much larger than the size of a cache line give better results for a search operation. Authors have proved that, when the node size is in a range of 1280-3072 bytes, the performance of the index can be improved up to 19% over the performance when the node size is 32 bytes**. Furthermore, they demonstrated that using the common heuristic of setting the node size equal to the cache line for this index structure is often suboptimal. Thus, in our researches we took a closer look at the correlation of the nodes size and the cache line in TPR-tree. We present the results in Section 6.

5.3 Complexity trends. Heuristics

Considering the last twenty years of researches in the branch of different data structures and access methods a mainstream is observed: the performance improvement beard by heavier CPU calculations. As an example, we investigate TRP-tree and its ancestors [14,18,19], literally R-tree and R*-tree. We notice that, each descendant is *heavier,* in the sense that more CPU calculations is required to build the tree. While the R-tree has only one minimization heuristic, its descendant, the R*-tree, uses four. Obviously, number of calculations required to minimize one heuristic is smaller then that required by four heuristics. However, the R*-tree performance is improved up to 50% compared with R-tree. The TPR-tree extends the R*-tree and according to that trend a greater number of calculations is required to minimize heuristics, since integral of functions use by R*-tree are computed. (see Section 2.3.3). We observe that trend for disk resident indexes, where node size is equal to the disk page, one I/O operation is generated every time a node is accessed. For such structures the number of I/O operations is the determiner of performance, because the time spent on the disk access is far greater than the time spent on CPU calculations. Thus, the most desired optimization is to limit the number of I/O operations. And naturally, that is mainly achieved by increasing the number of CPU calculations, insignificant when considering disk based structures.

However our researches focus on TPR-tree residing in main-memory, where I/O times are not the performance indicator. Since, the memory access time for RAM is measured in processor cycles, the complicated calculations come into the

* 44 bytes is a size of the entry in our implementation which is discussed in details in Section 6.

** 32 bytes is a size of a cache line in the experimental setup used in [10]

prominence. While one memory access takes up to 200 cycles, a calculation of some complicated integrals can take more than that.

Our idea is to investigate the situation with reversed trend. Which means that during our researches we will take under consideration following question. How much the performance of the TPR-tree suffers from reducing some complexity of calculations in heuristics? Especially, we will investigate that for deletion and insertion. According to our idea less CPU heavy algorithms may buy us more time, particularity in such a complex data structure as a TPR-tree where, for instance, an integral has to be counted on every level for every insert and delete operation. Simpler algorithm also mean less data needed for recognizing the situation– less data means less memory accesses. That means less time lost on waiting for the data to arrive.

As mentioned in the introduction section, dealing with spatio-temporal databases, application where updates are more frequent than queries are easy to find. Lets assume the simplicity of insert operation save one second for each operation. Multiplying that by the number of insertions performed in huge database is going to result in a significant amount of time. We need to remember that, for TPR-tree every update operation consists of a pair: delete and insert. By updating, we rebuild the tree and there is high probability that simpler insert operation produces less effective tree structure. That means the performance of the search operation drops down. Since the deletion uses search algorithm to localize the entry to be removed, the time of an update suffers. That leads as to the conclusion that the insert algorithm can be simplified as long as the update time decreases. However, the performance of select operation must not be forgotten also.

The TPR-tree, as a index structure, supports tree basic operations: insertion, deletion and searching. The main goal for optimizing the insert and delete algorithms is to design them in the way ensuring the best structure for searching. As mentioned above, that optimizing usually leads to more complicated CPU calculations, which are not important for a disc-based structure, but in our case can be one of the significant factor. We identify the heaviest parts, in terms of CPU cycles, for a update operations.

1. During insertion, when choosing an appropriate insertion path and when correcting of the tree structure after the insertion, Correction involves split and reinsertion algorithms. These situations require heuristic's penalty metrics calculations, which are integrals of functions of time. Following functions are considered: area of the bounding rectangles, their margin (during the split), the intersection of two bounding rectangles, and the distance between the centroid of an MBR (during reinserting). Those penalty metrics are used to
2. During deletion, when searching for the entry to be removed and when correcting the tree after underflow situation. While first situation involves the search algorithm, the other one involves the regular insert algorithm during the reinsertion.

During the experimental part of this project we will make an effort to take a closer look at the above conclusion.

5.4 Our Idea of Improvements

5.4.1 *Lazy Delete*

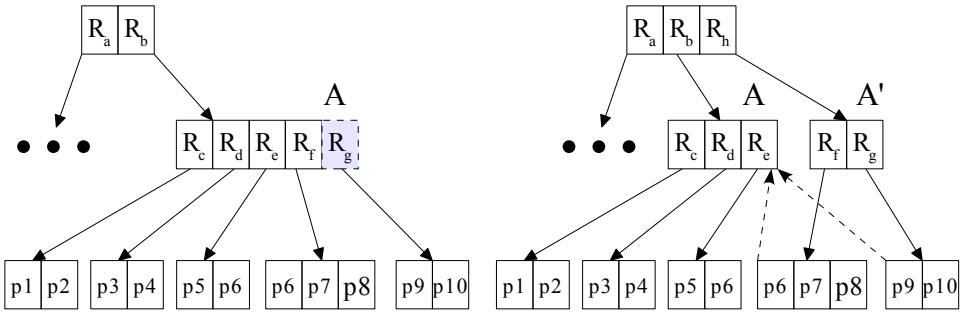
Analysing the delete algorithm of TPR-tree we can outline following components: (1) locating the entry to be removed, (2) removing the entry, (3) correcting the MBRs. In a situation where a leaf node after removing the entry is underflow a correction more complex, then just updating the MBRs on the upper levels, needs to be made. Underflow nodes are removed, the orphan entries are reinserted, and sometimes the tree is shorted. It is worth to investigate how to evade those heavy situations during the deletion. Since all data are kept in leaf nodes all underflow situations start on the leaf level and perhaps propagate up. Lets assume a change of underflow policy and remove the leaf nodes only when they become empty instead of less than half full. Notice that no more reinsertions of the entries from the leaf nodes will occur. For the internal nodes underflow rules remain unchanged. The crux of the new policy is not the improvement of every single delete operation but only the situations on the leaf level where the original TPR-tree underflow treatment would step in to the action. Furthermore the change of policy does not decrease the cost of single tree rebuilding after deletion, but minimizes the frequency of occurrence. We term the new policy as *lazy delete*.

5.4.2 Bottom-Up Delete

The other approach, unlike the one avoiding complex situations, applied in *lazy delete*, is to reduce the complexity of computations. Looking at the components of deletion operations one can locate the part where to adopt such a technique. The operation of removing an entry (2) is simple and can not be further simplified. Frequent corrections of the tree, performed during the update operations, guarantee the the high performance of search operations. Since that performance depends on the quality of corrections, simplification of algorithms could result in the the growth of time needed for the search operations. Motivated by the researches in [33] we employ a different approach of locating the entry in delete algorithm. It is worth to notice that searching of entry to be deleted is led by the objects position and the usually algorithm traverses more then one path down the tree. On the leaf level objects are verified by their unique Id. In the worst case the whole tree may be traversed and the last found entry may match the Id. Lets assume that we have an extra index table consisting of tuples (*np,Id*), where an *Id* represents the objects *Id* and *np* is a pointer to the node containing the object (see Figure 11).

The table is updated with every insert and delete operation. It allows to directly locate any given entry by unique *Id* with no tree searching. The introduced technique reduces the complexity of the delete algorithm with no decrease of search operations performance, because the corrections of the tree are performed as often and with the quality as in the original TPR-tree. The negative side effects of operating on two rather than on one indexing structure are overwhelmed by the benefits of more efficient delete algorithm. Following the naming convention uses in [33] we term this technique *bottom-up*.

The deletion in the TPR-tree involves two travels through the tree, one down to locate the entry and the other one up to correct the structure. Since the extra index table allows directly achieve the node containing the entry to be removed one travels is omitted. But still we need to go up the tree and for that purpose a parent pointer need to be added to each node. That add-on causes that maintenance of the tree is more expensive and split algorithm needs to be modified. The modified algorithm reads all the child nodes in newly created node in order to rewrite the information about their new parent in each one of them. Basically, during a split, a new node is added to the structure and half of the overfull node children are moved to the new



(a) Before the Split *(b) After the Split*

Figure 11: Additional Cost of Parent Pointer.

one. The parent pointers in children of the new node need to be updated. An example is presented in the Figure 12.

In the figure the maximum number of entries in a node is 4. A node, marked as *A*, is overfull and needs to be split. The situation after the split is presented on the Figure 12(b). New node have been added, and it is marked as *A′*. Without the modification of the algorithm the parent pointers in its children indicate the node *A,* (dashed arrows). After the modification all *A′* children need to be accessed to make their parent pointers valid.

In the Figure 12 only two updates of parent pointers are required. That number is equal *M/2* where *M* is the maximum number of entries in the node. Thus, with the growth of *M* the bear of having the parent pointers is greater. Since the node needs to keep an extra pointer, the other disadvantage for the in-memory data structures is a larger amount of space used.

5.4.3 New Penalty Insertion

In Section 5.3 we identified the complex parts of the insert algorithm. Since, to compute the penalty for a new entry to be inserted requires minimization of three heuristics to be calculated, we investigate how it can be simplified. We propose a different way to calculate the penalty by reducing the number of heuristic metrics in order to decrease the time spend on the calculations. Since that simplicity worsens the tree structure, we investigate the influence of *new penalty* on the search performance.

6 Experimental Evaluation

In this section we present the performance results of the index structure. The structure has been tested with combination of different sets of modifications. We present the test environment and the measured factors, followed by the description of the modifications and discussion of the results.

6.1 Experiments Settings

6.1.1 Test Configuration

In all our experiments we used a Pentium M 730 (Dothan) 1.6 GHz with 512 MB of memory. The processor has 64 KB of L1 cache memory with line size of 64 bytes. It consists of two 32 KB caches used for instructions and data. The 2048 KB of second level o cache are unified. All of the experimental implementations were compiled using GCC version and glibc version 2.4. We have not used applied any compiler optimization ("-O0" flag) to be able to use the debbuging of programs and in order to obtain results from the cache simulator (explained in section below) the closest to the reality*.

* The Callgrind may give false results when using optimization. We assume that the performance improvements are compiler optimization independent.

6.1.2 Measured Factors

Mainly, we have been measuring two factors :

- duration times
- cache performance.

Every time we mention of duration or time of a operation, experiment set time, we mean the number of processor cycles converted to wall clock time. We have used the read time-stamp counter (RDTSC) [28]. RDTSC is a set of registers build in to the processor counting all the ticks of the CPU from the system start. To convert the number of processor cycles to the wall clock time we use the following equation.

$$t = \frac{n}{f}$$

Where t denotes time in seconds, n – number of processor cycles, and f processor speed in Hz. I.e., the program execution is 216951259696 cycles on our machine it will take

$$t = \frac{216951259696}{1600000000\text{Hz}} \approx 135{,}5\,s$$

In some experiment, for certain proposes were comparing pure CPU cycles.

To measure the performance improvement of certain operation or total program time we use percentage improvement metric given by following Formula 1.

$$\frac{performance\,before - performance\,after}{performance\,before}\,\% \qquad (1)$$

The cache performance has been measured using Callgrind, a open source call-graph generating cache profiler [30]. Callgrind is a extension to cache simulator, Cachegrind witch is a part of Valgrind [31], a open source dynamic binary instrumentation framework. Cachegrind can simulate the behaviour of the cache on every level and give detailed performance statistics such as the number of cache reads, misses, hits of the program or separate functions. Also it can point a line in the code responsible for analysed rates so that it is easier for programmers to rethink the construction of hot parts of a code. Callgring, with its graphical frontend Kcachegring, can analyse the executed program (no simulation) and generate graph of calls between the functions.

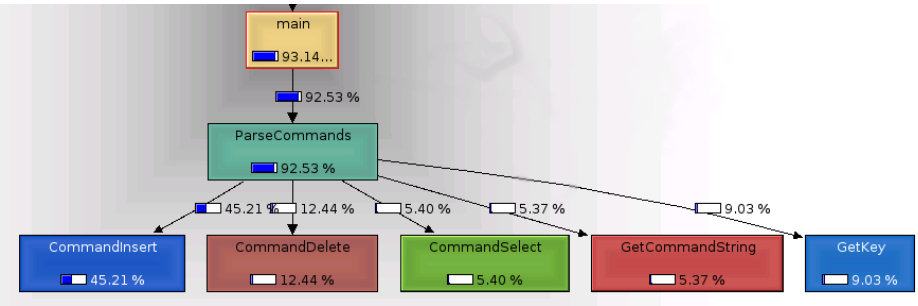


Figure 12: A sample Callgrid output

I.e. in our case the function *main* calls function *ParseCommand,* which calls for other functions, as presented in the Figure 13.

This, combined with the Cachegrind can visualize a path to a most of problematic code lines. For instance a small and simple function can be the source of all the problems on the second level of cache, because it is called with every operation on the node. So by having such path we can : reprogram the function and reduce the number of problems or reduce the number of calls to that function i.e. when this function cannot be corrected.

6.2 Generator and Workload

The workload is expected to simulate a wide range of situation in which the tree can be used. To create a workload for our experiment we use the generator developed originally for TPR-tree, by Saltenis et al. [14]. This generator is capable to intermix the queries with updates with a chosen proportion and a numerous parameters allows to generate any desired workload. In case of our workloads we use following parameters:

- *Space parameters.* In our experiments we simulate moving points (i.e. cars) in two dimensional space with are of 1 000 000 square kilometres. With the exception for a experiments where we investigate the scalability of the TPR-tree.
- *Number of moving points.* For most of experiments it is set to 100 000 objects with the exception for the scalability experiment where we use 10 000, 50 000, 100 000, 150 000, 200 000, 250 000 and 300 000 objects. For larger number of objects we scale the spatial dimensions of the space as recommended in [14]. The number of moving points is constant during one simulation, no new objects appear and no disappear.
- *Number of updates.* We set that parameters always as a multiplicity of the number of objects, that to update each object at least five times.
- *Number of destinations - ND.* The parameter determinates the skew of the

data and velocity. If it is set to 0, the generated data are uniformly distributed in a space. Mostly we use *ND*=20. That correspond to 380 one way routes. In the experiment where we investigate the influence of the number of destinations on the performance of the TPR-tree we use *ND* ={0, 2, 40, 160}.

- *Update frequency - UI.* The average update interval is 20 seconds in the most of the experiments. The real time interval between two successful updates is uniformly distributed between 0 and 40 seconds (*2UI*). However, we also run experiments to investigate the update frequently influence on the performance, where we vary that parameter.
- *Speeds*. The maximum speed of the point is chosen randomly from the set of maximum speeds. In all our experiment we use a set of speeds: 0.75, 1.5 and 3 km/min.
- *Query interval.* That parameters describe the query occurrence frequency measured in the number of insertions. We generate queries every 500 insert operations
- *Query quantity.* The number of queries generated in one query interval. In our experiments we produce 10 queries at one time. Thus, the total number of queries generated during one simulation depends on the number of insert operations.
- *Query size.* The spatial part of the query denoting percentage of the data space. Usually we set this query size to 0,1% of the area.

Generator produce workload with the timeslice, window and moving queries and allows to set parameters denoting the fraction of each type in the total number of queries. For all our workloads the queries listed above are generated with the probabilities 0.4, 0.3, 0.3 respectively. For more detail about generator we refer to [14]. The experiments investigating the *lazy delete*, the *bottom-up* delete and the new penalty improvements assume the standard workload, with default parameter values mentioned above. Different set of parameters is used when investigating the impact of *ND*, number of indexed points and update frequency, on the improvements. When we vary the number of objects, we use uniform data. In these cases appropriate comments are added in the experiment description. In the Section 6.7 where we studying the select characteristics, we briefly describe used workloads.

6.3 Implementations

As mentioned in Section 5 the performance of data structures in main memory depends on many factors. Having the platform, operating system and programming language determined only the way how the structure is implemented give us the influence on the performance.

Our implementation is based on the original TPR-tree implementation by Saltenis et al. [14] with some patches by Jing Wong making possible running it on

Linux platforms. All the examples used in the following section assume as a programming language C++ and came from our implementation.

6.3.1 Locality

Lets assume that the node does not contain actual data but pointers to them. Most probably the node with data enclosed by its pointers occupies more than one cache line because pointed data are probably located at different memory address. So each line is not going to be utilized in 100% and more than just the necessary number of calls to cache or main memory is going to be needed.

According to that nodes with all their content should be placed in area of memory, lying one next to each other. That placement policy of the data in main memory is a part of well-known cache conscious programming technique called locality [30].

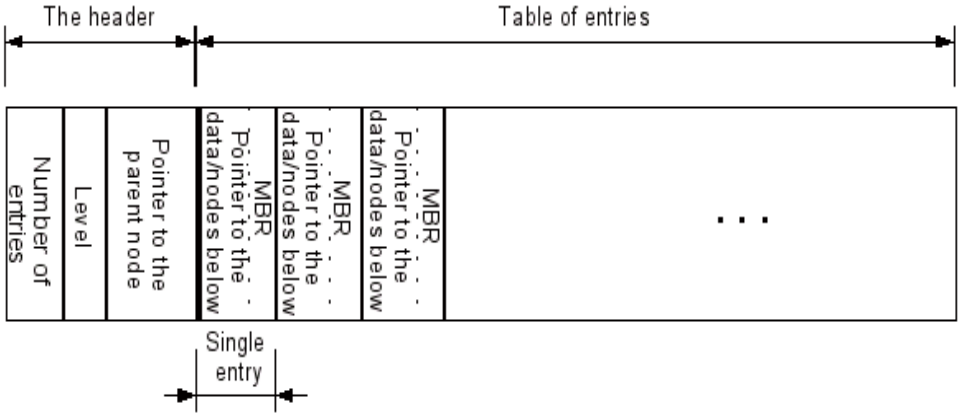


Figure 13: The TPR-tree node for main memory database

During the adaptation of the TPR-tree to new environment we designed the node following the rules of locality. The content of that structure can be theoretically divided to a part called a header and a table of entries as shown in the Figure 14. The header is where all the node parameters are located such as the level, number of entries etc. The table of entries stores pairs of Minimum Bounding Rectangle (MBR) and pointer to the child, when considering internal node or directly to the data – on the leaf level. This way when a part of memory is caches it is highly probable that it contains a full node, or at least a part without any additional unneeded data. This approach is a key to minimize the number of cache misses and thanks to that the number of reads from the memory paying back with less CPU time wasted.

6.3.2 Disadvantages of Obcject Oriented Programming

In order to move the implementation of TPR-tree from disk to the main memory we had to modify the structure to make it more compact. The original implementation was based on the Generalized Search Tree for Secondary Storage (GiST) [32]. This framework takes advantages of C++ language – uses object-orientation like inheritance and virtual functions. The next attribute of GiST is that pointers are commonly used. All the attributes of GiST we have presented can not be accepted for the index running in main memory. The idea of GiST was to support the secondary storage and not the main memory. The inheritance and pointers do not guarantee a coherent memory map which prevents creation of compact data structure. The virtual functions are less effective then regular functions. They need more time because the polymorphic call is required. Moreover, they use more space there needs to be a virtual table for each class that has a virtual method [31]. All that motivated us to code our implementation without using GiST.

6.3.3 Space Utilization

When it goes for the main memory resident indexes the size matters. The redundancy reduction of data in the structure is needed. I.e. our tree keeps the information of the level only in the node while the original implementation stored it in the node and in each entry. Since the entries are the basic component of the nodes, a simple improvement could save the great amount of the memory space. Lets assume that a node has 10 entries and the information about the level is stored as integer in each entry. In each entry that information, keep as a integer, occupies 4 bytes. Removing it from entries to the node 36 bytes are freed (4 bytes need to be kept in the node).

Another approach for saving a space is replacement large, complex structures with more simple ones. In our of adaptation, such a technique mentioned in Section 5 is replacement of *path* by simple parent pointer. Originally in a GiST *path* is a table of pointers to the node, where a sequence of nodes' pointers from root to the certain node is kept. While for a search proposes MBRs and pointers to the child are used, *path* is used for traversing up. There is no algorithm in the TPR-tree where a node directly calls another node lying more then one level up. The only exception is a call to the root which is handled in a different way. Summarizing removing the *path* saves space with no reduction of functionality but effects in more complex algorithms. (See Section 5).

More space can be bought by responsible data type usage. Variables are represented by different data types, which occupy different size of memory. There is no sense in overestimating the range of stored data. I.e. in all of our experiments we never had tree higher than 20 levels, but the original implementation was using four bytes (integer) for storing that information. One byte can store up to 256

combinations, so a char type store the level is enough. Similar situation evolves variable storing number of entries in the node – for 512 entries 2 bytes are enough. An unsigned short is enough and there is no need of using the twice larger integer.

6.3.4 Functions

Sensible way of implementing the functions is competition of the data structure. We have observed that some parts of code implemented in similar ways can result different cache utilization and can cause a significant number of cache misses. A simple example which we had in our work was that object creation can very expensive.

```
1     for (int j=0; j<numEntries; j++)
2   {
3     RTnode node = *this;
4                                   if (query.Consistent(node,j))
5     {
6       return j;
7     }
8   }
```

(a) Before

```
1     for (int j=0; j<numEntries; j++)
2   {
3       RTentry* e = (RTentry *)&entries[j];
4     if (query.Consistent(e,this->IsLeaf()))
5     {
6       return j;
7     }
8   }
```

(b) After

Program listing

Creating a new object (Program listing (a) line 3) as a temporary copy of existing one will result in high load of cache because all the data needs to be read from the original and written to the copy. In functions that are often called that may result in great number of memory accesses and cache misses. If possible it is better to use the pointers to the current object ("this" - Program listing (b) line 4) or pointer copies of original objects (Program listing (b) line 3). The small difference we show in those two program listings -modification to the way function is called saved us 30% of the global number of L2 cache misses.

It is important to identify the top called functions during the program execution. These functions which may be very small may take most of program execution time. That is why they need to be optimized. Each error costs more than in function called only few times.

6.4 Main Bottelneck

In the first experiment we vary the node size and measure the times of the global execution and the operations' times. We measure the time spent on insert, delete and select operations. The we investigate what is the main bottleneck of TPR-tree running in main memory is. We measure the percentage of time spent on cache misses and the structure of those misses.

The Figure 15 presents the performance of the TPR-tree varies with the node size. The measures starts with the minimum reasonable number of entries, which is four when node size is equal to 236 bytes. On he Figure 10(a) the node size growths by ten entries to the maximum of 100 (4020 bytes). The Figure 10(b) is more detailed and varies by every two entries up to 30 entries (1380 bytes). In general the times rise with the node growth as well for smaller node sizes as for greater ones. The only exception is for the smallest node 236 (carrying 4 entries). The global time of a program running exactly with the same workload, for a node size 1380 bytes is almost 2 times longer then the time spend with node size 236.

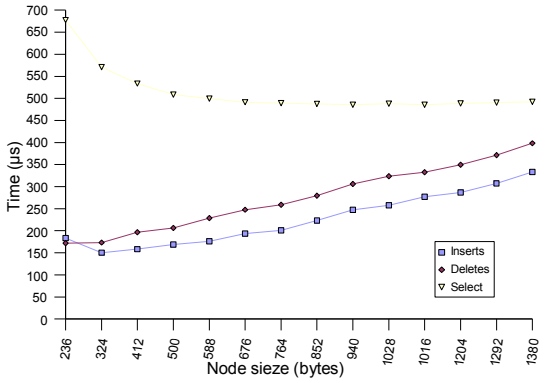


Figure 14: The performance of operations of TPR-tree

The Figure 16 shows the dependency of single operation time varying form the node size. The performance trend of update operation is opposite to select operations. While the time of the single select operation sightly decreases with the growing node size a single update operation consisting of deletion and insertion rises. It is worth to notice that optimal node for update performance is the same as for the global performance of the TPR-tree.

Next we investigate the role of cache misses in main memory TPR-tree. We measure the total number of cache misses in the program. Then we estimate the time spent on memory access and the CPU calculations. Notice that the results are the estimations but always rounded to the worst case values. The cache miss penalties

are not constant even on the same hardware, operation system and can vary by every experiment. In our setup we have assumed a penalty of 20 cycles on the fist level of cache and 200 cycles on the second level. Probably such bad factors are never reached. The tests were run for three representative node sizes : the smallest one (236 bytes) , the one with the best performance (324 bytes) and one illustrating the decreasing performance (1380 bytes). Results are shown in the Table 1.

Node size	236	324	1380
Estimated global time spend on cache misses	6 421 820 620	5 326 742 200	3 223 677 540
Measured execution time	274 189 899 957	248 699 909 561	541 201 704 506
Percentage of time spent on CPU calculations	97,71%	97,90%	99,41%
Percentage of time spent on cache misses	2,29%	2,10%	0,59%

Table 1: The cache measurements and time estimations

The cache misses take only a small fraction of the execution time, reaching not more then 3% . For the smallest node size 2,29% of time is spent on the direct reads from memory. That rate drops down with the node size growth. While for the optimal node size it is equal to 2,10%, for the larger node it is only 0,59%.

The results obtained in our experiments indicate that the TPR-tree is diferent from the structures that were taken under consideration in experiments evolving main memory data structures. Previous researches have proved that the most significant part of execution time is spend on second level data cache misses and first level instruction fetch misses [3]. Although that time spend on cache misses in our experiment is insignificant we check if the structure is similar. To investigate that we take a closer look at the structure of the measured cache misses as shown in the Figure 17.

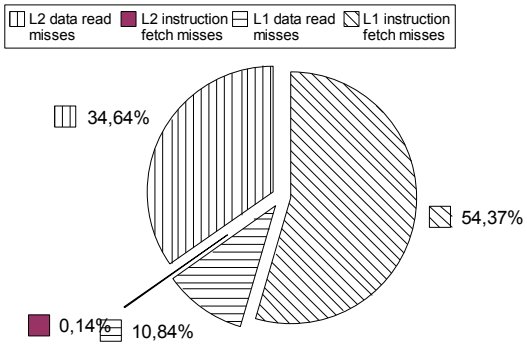


Figure 15: The time structure of cache misses.

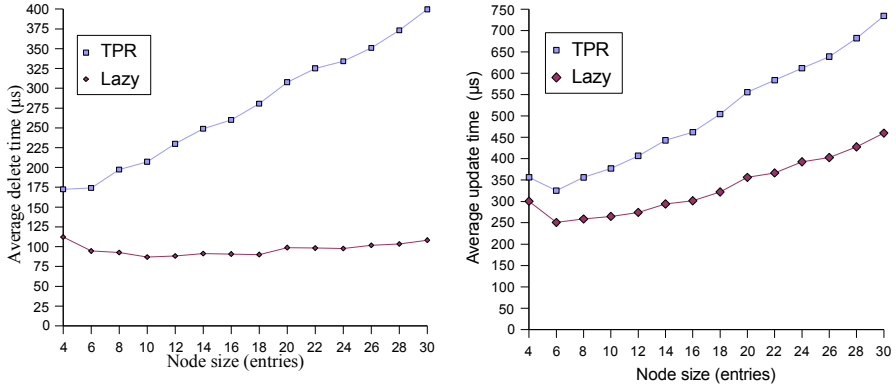
The Figure 17 presents the time structure of cache misses for the size 236 bytes. This node resulted with the highest number of cache misses - 2,29% of the global program time. That number consists mostly of instruction fetch misses on the first level. The next key factor is the time spent on the L2 data read misses. And only a small amount of the time is spent on L1 data read misses while the L2 instruction fetch misses are insignificant.

The above results lead to the conclusion that for TPR-tree the cache misses are not a main factor influencing on its performance. The amount of cache misses is small, their structure is consistent with other main memory strictures [3]. Since more than 98% of program execution time is spent on CPU calculations in next experiments we concentrate on simplifying the algorithms. Considering the update and looking at the Figure 15 we choose to improve first the delete as a more time consuming algorithm.

6.5 Delete Improvements

6.5.1 Lazy Delete

Following the investigations from Section 5 we implement the *lazy delete* policy in to the TPR-tree. The following experiments compare the new policy with the original TPR-tree deletion. We vary the node capacity from 4 to 30 entries, which reflects the sizes from 236 to 1380 bytes. Figure 17(a) and (b) shown the measured elapsed time per deletion and update operation respectively.



(a) Delete Time *(b) Update Time*

Figure 16: Lazy Delete Performance

As we can observe in the Figure 17 the influence of *lazy delete* on the tree performance is significant. In case of optimal node, obtained in previous experiment, where capacity equals 6 entries (324 bytes) the improvement of delete time is about 45%. While the average update time is over-performed by 23%. Looking at the Figure 17(b) we can observe that the optimal node size has not change in term of update duration, but for deletion (Figure 17(a)) it has been switched to node of capacity 10 entries (500 bytes). However, for *lazy delete* the optimal node size is not so obvious any more. Average delete times for all node
sizes presented in the Figure 17(a) do not differ one from the other more then 8%. Moreover the node size is no longer the key factor of the delete operation performance.

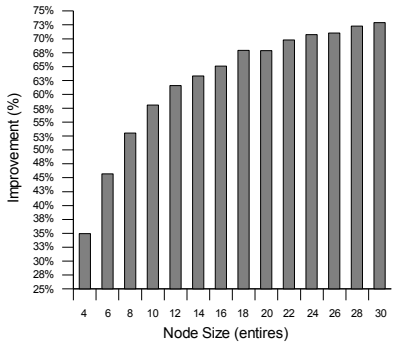


Figure 17: Delete Improvement by Lazy Delete Policy

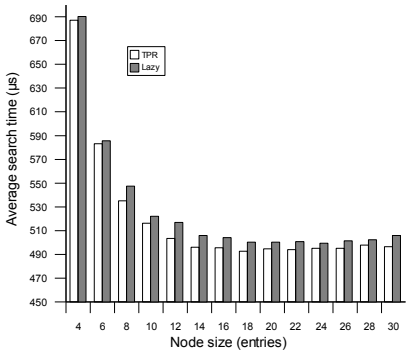


Figure 18: Influence of Lazy Delete on the Search Performance

The tendency where delete time increase with node growth does not appear when *lazy delete* is used. Furthermore as shown on Figure 18 the improvement caused by *lazy delete* increases as the node size grows. While for the smallest size of the node it is about 35%, for the node of capacity 30 entries it raises to up to 70%. This can be explained in the following way.

The main change made by *lazy delete* lies in avoiding reinsertion of entries from underflow nodes. Considering the TPR-tree with original algorithms and maximum capacity of node 4 entries the underflow leaf nodes causes reinsertion of only one entry. In the same situation where node can hold 30 entries, 14 reinsertion take place. When applying *lazy delete* for a trees with small nodes it does not reduce the number of reinsertion significantly. In the Figures 17 and 18 we can observe that *lazy delete* improves the average time of delete for a small node sizes, but for a bigger nodes the improvement is much more noticeable. To investigate that more carefully we measure how many times the underflow situation occurs for both policies and then we calculate the percentage of avoided ones. We run these experiments for the node capacity of 4 and 30 entries. The results we obtained show that for the smallest node size about 20% of underflow situation are avoided by *lazy delete* and that percentage grows to more then 65 for the other node size.

We investigate what cost the search operation bears for the update improvement caused by *lazy delete*. Figure 19 presents the influence of *lazy delete* on the search performance. While the betterment of update and delete is undisputed, the negative influence on the select is hardly noticeable. The average select time in the original TPR-tree is never better by more then 3%. This slightly degradation can be simply explain. Applying *lazy delete* policy causes that for the same amount of kept data and the same node capacity the number of internal nodes and entries in the tree is greater. For search operation that means that more nodes and more entries in these nodes need to be checked during one select operation. Each entry is checked if its MBR satisfies the query to move down the tree with that path. Figure 20 presents the average number of visited nodes during one search operation. That number is slightly greater for a tree with *lazy delete* applied.

The result of the researches presented above were obtained from with one simulation workload. The parameter that varied was the node size. We proceed to describe how the *lazy delete* behaves in different environment simulated by the workload. We investigate the influence of skewed and uniform data, update frequency and number of indexed points on the tree performance. All those experiments were run for a node size of 12 entires.

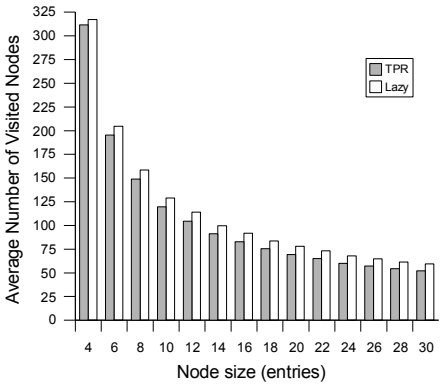


Figure 19: Search Performance

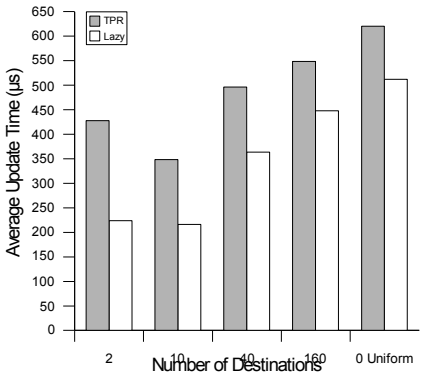


Figure 20: Average Update Time for Varying Number of Destinations

Figure 21 presents the results of the experiments run with the workloads where the parameter *ND* (number of destinations) was varied. The influence of number of destinations of the average update times is shown. As Saltenis et al. in [14] show that uniform distribution of the object positions and velocity vectors is the worst case. The more skewed data are, the assigning them to to bounding rectangles with small velocity extents, is easier. In our experiments we observe, that this trend remains unchanged for TPR-tree with applied *lazy delete* policy.

In the TPR-tree during every update the MBRs are tightened. Thus, the more frequent updates lead to higher efficiency of the structure. With the update interval relatively long the MBRs grow in size increasing the overlap area drastically. As presented in the Figure 22 the average *lazy delete* time is smaller then the original. However gap decreases with updates occurring less frequent.

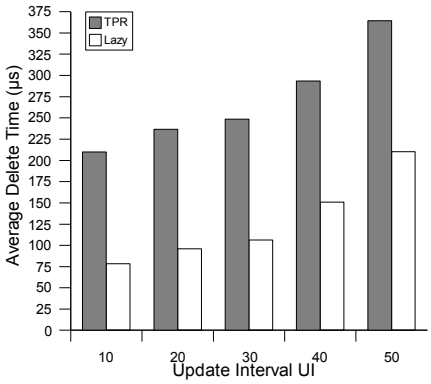


Figure 21: Average Delete Time for Varying UI

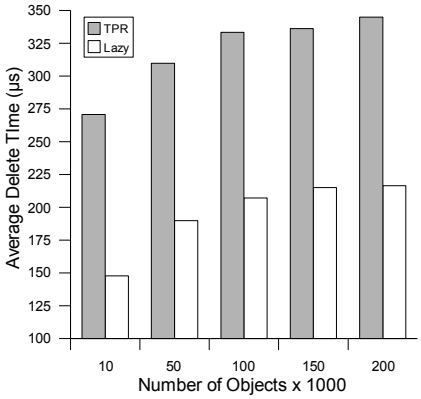
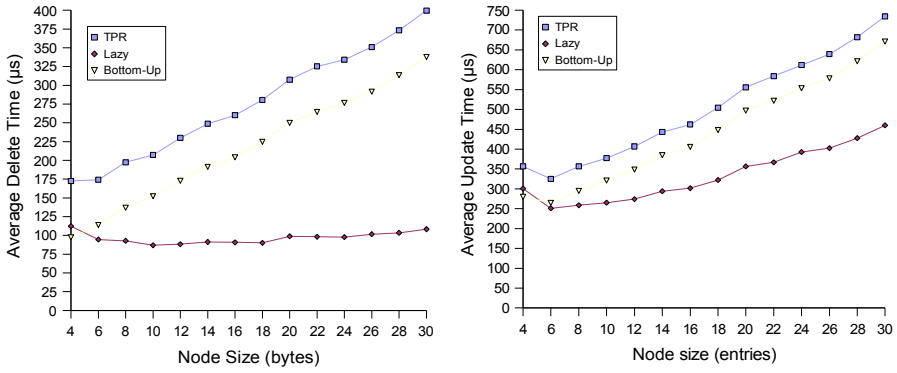


Figure 22: Average Delete Time for Varying Number of Indexed Points

As shown in the Figure 23, when varying the number of objects, the average delete times for *lazy* delete approach and original TPR-tree differ by almost the same interval of time.

6.5.2 Bottom-Up Delete

Other approach (mentioned in Section 5) - the *bottom-up* technique is expected to improve deletion simultaneously with no decreasing the search performance. The following experiment verifies that statement. The measured factors remain the same as in previous experiments. Figure 23(a) depicts relations between the average *bottom-up delete* time with varying node size.



(a)Delete *(b)Update*

Figure 23: Bottom-Up Improvement

Comparing the average time of *bottom-up delete* with the original TPR-tree in the same case as done in the previous experiment (optimal node size) we have obtained the results as follows. The delete operations are improved by 19% and updates by 35% per every operation. But as presented in the Figures 24(a) and (b) and in contrast with the *lazy delete* policy the improvement depends on the node size. Furthermore it reacts on the change of the node capacity in the same way as the original delete, it can be observed in the Figure 23, where the TPR-tree curve is almost parallel to *bottom-up* curve. With the exclusion of the smallest node the results differ with constant interval of time. That is the time spent in original TPR-tree on the searching the entry to be removed. The exception can be easily explained – the TPR-tree height with node capacity 4 is drastically higher than for the capacity of 6. In case of smallest node (236 bytes) the tree has 11 levels. The growth of node to 324 bytes (two more entry in the node) causes reduction of tree height by 3 levels, while for further growth of node such a great reduction is not observed. For delete algorithm searching the entry to be removed is more expensive in higher trees because number of nodes to be checked in a path form the root to the leaf grows rapidly. Since applying *bottom-up* approach eliminates searching from the deletion, the minimum average time is reached for node of capacity 4 entries. For the update the optimal node size is not changed compared with the TPR-tree - it is still caries 6 entries.

Our experiments confirmed that *bottom-up* does not have any influence on the select operations. The results we obtained show that the average select times, for the

original and *bottom-up* approaches, differ from less the 0,5% to 2%. This small differences may be cause by way operating system works. It is worth to notice that even two identical experiments run one after another in the same environment may differ. However all remaining indicators of the tree structure like the tree hight, the number of internal and leaf nodes, average fill of nodes and average size of MBRs on each level are equal for the TPR-tree with and without *bottom-up*.

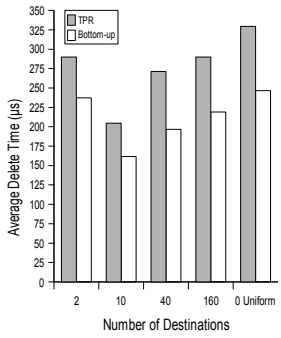


Figure 24: Average Delete Time for Varying Number of Destinations

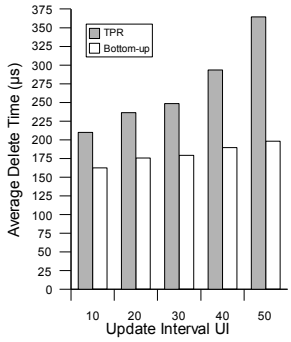


Figure 25: Average Delete Time for Varying UI

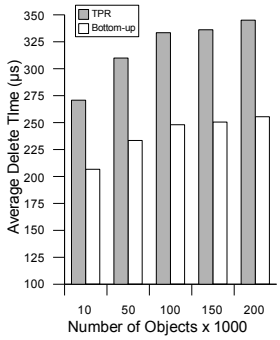


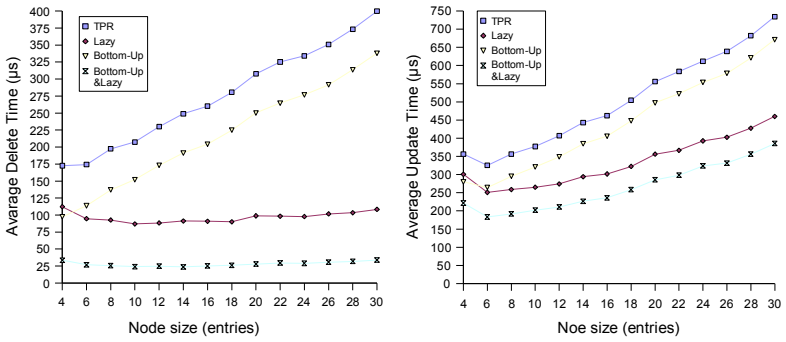
Figure 26: Average Delete Time with Varying Number of Indexed Points

The following experiments were run for a node size of 12 entries. Experiment for the varied number of destinations shown (see Figure 25) that *bottom-up delete,* as expected, decrease the average delete time for TPR-tree by a constant value in each case. When the update interval is varied, average *bottom-up delete* time does not grow so fast as TPR delete time does, as it is observed in the Figure 26. That is natural, since *bottom-up* does not involve any searching and increase of update interval causes the overlap enlargement. Similarly, like for *the lazy delete,* the improvement archived by *the bottom-up* does not depend on the number of indexed objects.

6.5.3 Lazy and Bottom-Up Delete

The Figure 28 shows the comparison of two the introduced techniques as well the combination of them with the original TPR-tree. The new delete techniques refer to different parts of the algorithm they can be applied simultaneously. Thanks to that the delete becomes almost node size independent at least for the nodes sizes examined in our researches. That is the result of the *lazy delete* since the previous experiments show that the *bottom-up* is responsible only for the parallel down displacement of the graph (see Figure 24). The best results are obtained for the combination of *lazy* and *bottom-up* for the deletion as well as for the update

performance. In the worst case (the smallest node size) the average time of delete is improved by 80% and grows for a largest node size measured to 90%. Hence the size still influences on the the average update times, with the growth of the node size performance decrease.



(a) Delete Time *(b) Update Time*

Figure 27: Delete Improvements

Since both delete improvements separately show no significant influence on the select and there are no theoretical reasons for worsening the performance by their combination we do not discuss that.

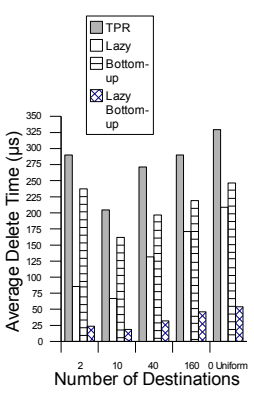


Figure 28: Average Delete Time for Varying Number of Destinations

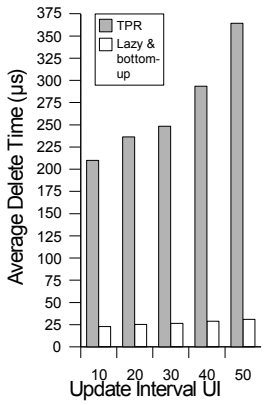


Figure 29: Average Delete Time for Varying UI

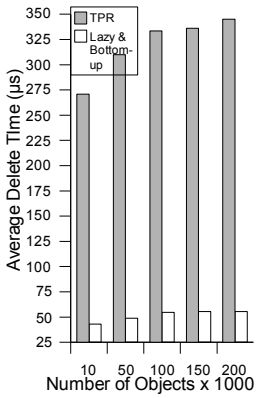


Figure 30: Average Delete Time for Varying Number of Indexed Points

However, we investigate both improvements applied in the same workloads that used in earlier experiments. The combination of *lazy and bottom-up delete* does not yield any unexpected results. The Figeres 29, 30, 31 present results of our experiments for varied number of destinations, update interval and number of

indexed points. In the next experiment we investigate the insert algorithm as a significant component of update operation. All of our experiments ran for *lazy and bottom-up* combination proved that, independently of simulated situation, the archived improvement is significant.

6.6 Insert Improvement

Looking at the results of our earlier experiments it becomes clear that the delete algorithm can be easily improved with no sufficient worsening of the select performance. According to the results obtained in the first experiment the most of the time in the TPR-tree structure is spent on the CPU calculations while the memory accesses is not the key factor to the general improvement. In this experiment we follow the divagations about complexity from Section 5 and simplify the insertion algorithm showing that the select operation are not going to suffer.

We check our implementation using the Callgrind program for functions occupying the most of execution time. The same representative node sizes were tested as in the previous experiments (4,6,30 entries in the node). Table 2 presents results obtained from the Callgrind. Notice that in all presented cases the insert function takes about 50% of general execution time. The significant part of that time is spend on calculations employed by TPR-tree heuristics, since they are based on integrals (for details see Section 2). The heaviest of those heuristic, in term of CPU time spent, is the one computing the integral of the intersection of two time-parametrized rectangles and it is included in the Table 2. This algorithm is extension of select algorithm which check if those two rectangles overlap [34]. The insertion uses this heuristic during penalty calculation for determining the position for a new entry.

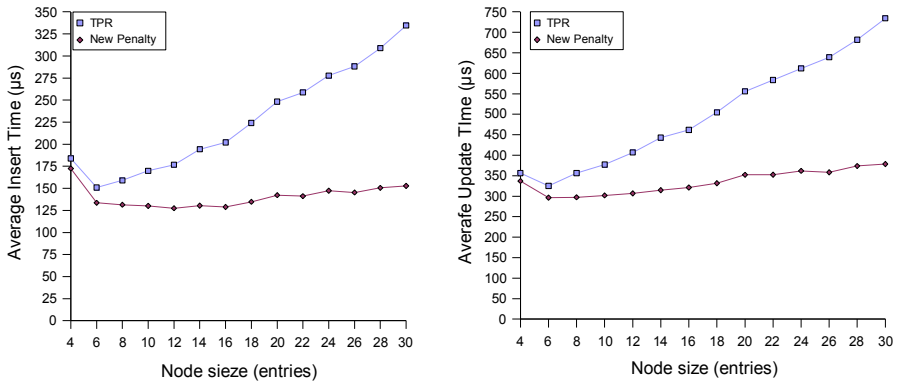
	Node Size (entries)		
	4	6	30
Percentage of program time taken by Insert	51,1	46,43	49,97
Percentage of program time taken by the integral of intersect area calculation	3,52	7,22	34,38
Number of overlap checked	3 096 136	4 016 206	10 725 690

Table 2: Results of Callgrind Analysis.

Looking at the Table 2 we observe that fraction of total program time taken by the integral of intersection area calculation is greater when the node size is greater. While for a node of 4 entries it is only 3,52 % , for a node of 30 entries it

shoots up to 34,38%. Moreover the number of calls for that function is more then three times greater for the largest node compared with the smallest one.

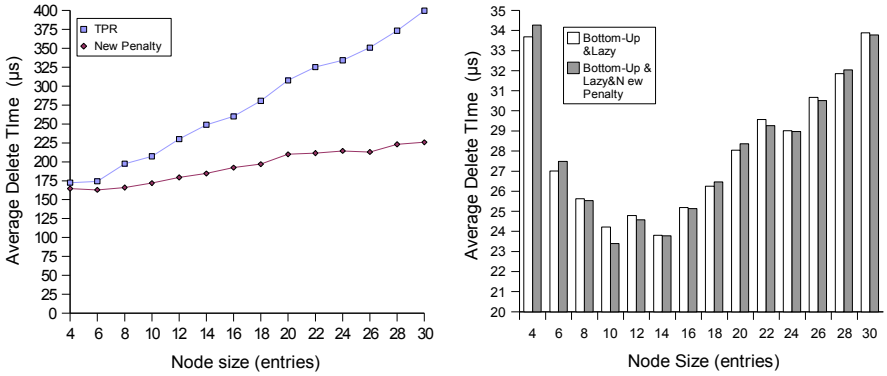
In the following experiment we check how omitting of the heaviest heuristic influences on the TPR-tree performance including insertion, deletion and select operation. We term insert algorithm without the intersection calculations, *new penalty insertion*. The Figure 32(a) and (b) shows two curves, one for the original TPR-tree and the other one for a *new penalty insertion*. Similarly to the *lazy delete* case, the average time of *new penalty insertion* does not depend significantly on the node size. Therefore the optimal node sizes, giving similar times for the insertion, are between 6 and 18 entries. The improvement of insertion for a node of 6 entries is only 6%, but grows with the node size to reach more than 50% for 30 entries in a node. Explanation for that is partly included in the Table 2, where we present the number of calls, to that expensive heuristic, avoided in the case of *new penalty* insertion. Since that number is greater for the greater nodes the better improvement for these nodes is obvious and can be observed in the Figure 32.



(a) Insert *(b) Update*

Figure 31:New Penalty Improvement

Deletion algorithm employs the regular insertion for the reinsertion proposes, thus each insertion improvement is expected to have positive influence on the delete performance. The next figure presents the influence of the *new penalty insertion* on the original TPR-tree deletion (Figure 33(a)), combination of *lazy* and *bottom-up delete* (Figure 33(b)).



(a) TPR-tree vs. New Penalty *(b)Improved Delete vs. New Penalty*

Figure 32: New Penalty Influence on Deletion

As shown in the Figure 33(a) the influence of *new penalty insertion* on the average delete time is significant. The improvement for a node sizes 4 and 6 entries is is minimal, only about 5%. With the increase of the node size, the improvement grows too. That occurrence is in accordance with previous experiments. During the delete operation reinsert occurs every time a node underflows. For nodes of little size the number of entries to be reinserted is smaller. Moreover the insertion for them is relatively cheaper comparing i.e. with the node of sizes 20 and 30 entries. Thus, that tendency has an influence on the average delete time. The distance between graphs for a TPR-tree and for a *new penalty* (see Figure 33(a)) reflects the time which is spent on the reinsertion by the delete algorithm.

In spite of expected betterment for combination of *lazy delete* and *bottom-up* improvements caused by *new penalty insertion*, it has not been confirmed in the experiments. This can be observed in the Figure 33(b). We explain that in following way. Since the *lazy delete* avoid most of reinsertions the improvement of insert looses the impact on the deletion.

Omitting one of the penalty metrics, what is applied in the *new penalty* insertion, should effect with worst tree produced. But, as presented in the Figure 34, the results comparing the influence of TPR-tree and *new penalty insertions* on the search times do not differ in a significant way. Surprisingly the graphs do not show any significant differences, but these results correlate with the structure indicators, like the number of internal entries, margin, or the the number of visited nodes (see Figure 35), not differing in significant way.

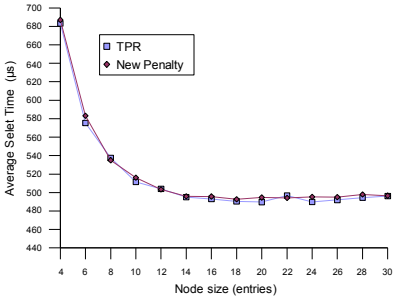


Figure 33: The New Penalty Influence on the Select

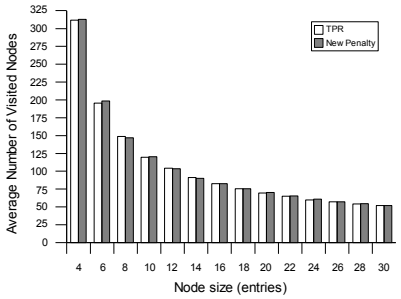


Figure 34: Number of Visited Nodes During Search Operation

Summarized, the update improvements do not decrease the performance of select operation significantly.

We proceed to investigate how the *new penalty insertion* behaves when differing the workloads parameters for the node size of 12 entries. For more skewed data the gap in average insert time between the TRP-tree and *new penalty insertion* decreases, while for uniform data it grows significantly (see Figute 36). The experiments for varied update interval (see Figure 37) show that, however the *new penalty insertion* lessens the time by a constant value, for both presented cases, average times grow steadily. Like in the case of delete improvements, *new penalty*, for different number of points the characteristics remain unchanged.

6.7 Studying Select Characteristics

In this section we generally investigate the negative influence of our improvements on the select characteristics. In the first experiment we compare the average times of select for the combination of all our improvements and the original TPR-tree. The default parameters for workload were uses in this experiment. As expected, TPR-tree over-performs the modified version for all examined node sizes. However the difference is small and does not reach more than 4% (see Figure 39). That is a result of the larger number of nodes visited during searching. In the Figure 40 the comparison of this characteristic is presented. On average 10 more nodes are needed to locate the same objects, required by the query in two different data

structures produced. This is caused by *new penalty* and the *lazy delete.* While the *new penalty* leads to area overlap enlargement, the *lazy delete* increases the number of leaf nodes in the tree.

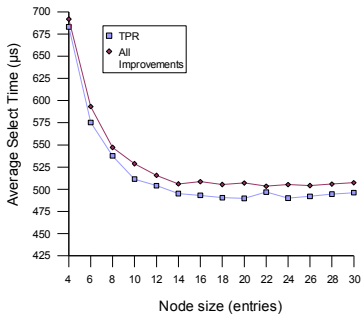


Figure 35: Average Select Time

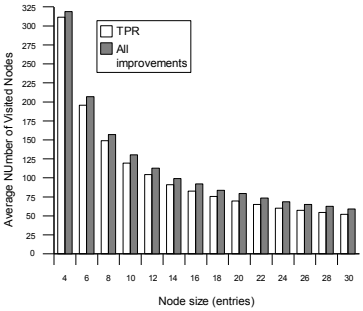


Figure 36: Average Number of Visited Nodes during Search Operation

Next we investigate if the worsen select performance is resulted also during different simulations. As a node size for that experiment we choose 12 entries as a representative, for the range, where the greatest influence of our improvements is observed. Figures 41, 42, 43 depict obtained results when varying a the level of skew of moving objects distribution in the space, the update frequency and the number of objects.

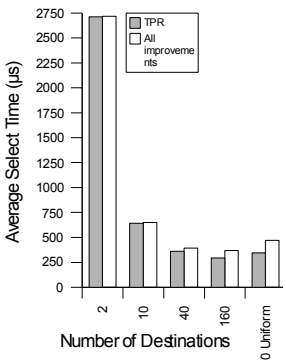


Figure 37: Average Select Time for Varying Number of Destinations

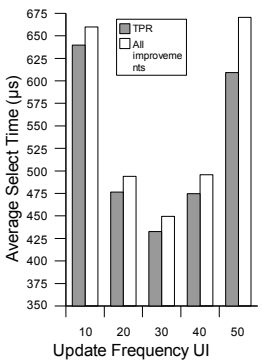


Figure 38: Average Select Time with Varying UI

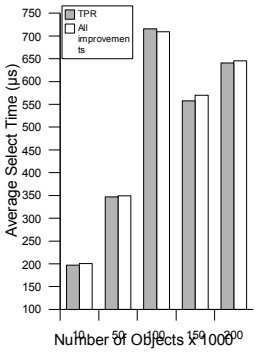


Figure 39: Average Select Time with Varying Number of Indexed Points

In all of these situations, the original TPR-tree answer the queries faster, as it is better adapted to the varying conditions than our simplified version.

In the following experiment we compare search performance of original, and the TPR-tree with *lazy delete, bottom-up* and *new penalty* applied simultaneously. We change the size of the query rectangles from 0.01% of the space to the 5%. The other characteristics of the workload remain as for a default settings, with the exception for a node size, which is 12 entries. Results of this experiment confirmed that TPR- tree needs less time to proceed a query operation independently of query spatial size.

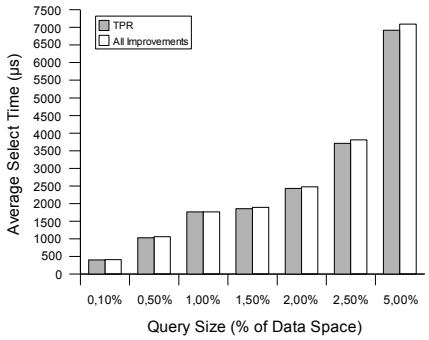


Figure 40: Average Select Time For Varying Query SIze

7 Conclusions

In this paper we have made the first attempt of analysing the Time Parametrized R-tree running in main-memory environment. We have verified that the growing gap between the memory and CPU speeds is not the main bottleneck in case of such complicated data structure. In contrast to that we prove that more than 90 % of time is spent on CPU calculations. Following these conclusions we propose three approaches of limiting the number of computations needed for the update operations: *lazy delete* – limiting the number of occurrences of heavily algorithms correcting the tree (underflow treatment, reinsertions and shortening the tree), the *bottom up delete* – where by using the additional indexing table we omit time-taking entry localization during the deletion. The third approach follows the ideas of earlier improvements and was implemented with help of binary program analyser - the Callgring. The program helped us to locate the heaviest of the heuristics used by the insertion – the overlap enlargement. We have modified the insert algorithm so that it omits this penalty metrics. As all of these improvements are applied in different parts of the algorithms there is no limitation of joining their performance.

In our researches we improved performance of update. However that improvement has just slight but negative effect on the query operations time. We have proved that the query operations suffer from our improvments, however we show that the decrease of the performance is less than 4%.

8 Future Work

We have presented that the TPR-tree running as a data structure designed for the disk oriented environment can be easily over-performed by simple modifications. As in the theory the CPU heavy algorithms were ensuring the performance by limiting the number of disk I/O the main memory environment they become the main bottleneck. The tree characteristics change so that the tree is becoming horizontally expensive – the relative cost of visiting an additional node drops down to values comparable to CPU cycles, while the number of computations performed on one node remains the same. Visiting another node takes less time than examining one entry. It would be interesting to rethink the number of penalty metrics needed for the update operations as well as the number of CPU heavy calculations involved in these metrics.

References

[1] W. Kim, W. Loh, W. Han. CC-GiST: Cache Conscious- Generalized Search Tree for Supporting Various Fast Intelligent Applications. ISI pages 657-658, 2006.

[2] J. Rao and K. Ross. Making B+-trees cache conscious in main memory. In Proc. of the ACM SIGMOD Conference, pages 475-486, 2000.

[3] A. Ailamaki, D.J. DeWitt, M.D. Hill, and D.A. Woo. DBMS on a Modern Processor: Where Does Time Go? Proc. 25th Very Large Databases Conf., pages 266-277, 1999.

[4] P. Bohannon, P. Mcllroy, and R. Rastogi. Main-Memory Index Structures with Fixed-Size Partial Keys Proc. ACM SIGMOD, pages 163-174, 2001.

[5] K. Kim, S. K. Cha, and K. Kwon. Optimizing multidimensional index trees for main memory access. In Proc.of the ACMSIGMOD Conference, pages 139-150, 2001.

[6] E. Shim, S. Song, Y. Min, J. Yoo. An Efficient Cache Conscious Multi-dimensional Index Structure. ICCSA (4) pages 869-876. 2004.

[7] I. Sitzmann and P.J. Stuckey. Compacting discriminator information for spatial trees. In Proceedings of the Thirteenth Australasian Database Conference pp.167-176 Melbourne, January/February 2002.

[8] D. Comer. The ubiquitous B-Tree. Computing Surveys, 11: pp. 121—137, 1979.

[9] Kyung-Chang Kim, Suk-Woo Yun. MR-Tree: A Cache-Conscious Main Memory Spatial Index Structure for Mobile GIS. W2GIS 2004. pages 167-180

[10] R. A. Hankins and J. M. Patel. Effect of Node Size on the Performance of Cache-Conscious B+-trees. In Proc. of the ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems, June 10-14, 2003, San Diego, California, pp. 283-294.

[11] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal Prefetching B+-Trees: Optimizing Both Cache and Disk Performance. Technical Report CMU-CS-02-115, School of Computer Science, Carnegie Mellon University, Mar. 2002.

[12] Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. Proceedings of Very

Large Database Conference (VLDB). 2003. pages 790-801.

[13] 0. Wolfson, B. Xu, S. Chamberlain, L. Jiang. Moving Objects Databases: Issues and Solutions In Proceedings of the 10th International Conference on Scientific and Statistical Database Management. Capri, Italy, July 1998.

[14] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In Proc. ACM SIGMOD International Conference on Management of Data, pages 331-342, 2000.

[15] G. Kollios, D. Gunopulos, and V. J. Tsotras. On Indexing Moving Objects. In Proceedings of ACM Symp. on Principles of Database Systems, 1999.

[16] Blewitt, G. Basics of the GPS technique: observation equations. Geodetic Applications of GPS (1997)
pages 10-54, 1997

[17] Civilis, A., Jensen, C.S., J. Nenortait˙e, J., Pakalnis, S. Efficient tracking of moving objects with precision guarantees. In Proc. 1st Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services, pages 164-173, 2004, extended version available as DB-TR-5, Dept. of Computer Science, Aalborg Univ., Denmark, http://www.cs.aau.dk/DBTR/DBPublications/DBTR-5.pdf.

[18] Beckmann, N., Kriegel, H., Schneider, R., Seeger, B. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. SIGMOD, 1990.

[19] A. Guttman. R-Trees: A Dynamic Structure for Spatial Searching. In: SIGMOD 84, Proceedings of Annual Meeting, pages 47-57. Boston, 1984.

[20] The Power Protection Handbook. Technical report, American Power Conversion, 1996.

[21] P. Bernstein, M. Brodie, S. Ceri, D. DeWitt, M. Franklin, H. Garcia-Molina, J. Gray, J. Held, J. Hellerstein, H. V. Jagadish, M. Lesk, D. Maier, J. Naughton, H. Pirahesh, M. Stonebraker, and J. Ullman. “The Asilomar Report on Database Research”. SIGMOD Record , 1998

[22] Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th VLDB Conference*, 1999

[23] NR Mahapatra and B. Venkatrao. The Processor-Memory Bottleneck: Problems and Solutions ACM Crossroads, 1999.

[24] Ruud van der Pas. Memory Hierarchy in Cache-Based Systems, Sun Microsystems , 2002

[25] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers , University of Virginia

[26] C. A. Shaer. Data Structures and File Processing, Virginia Tech ,1998

[27] J. Rao, K. A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. , VLDB Conference, 1999

[28] A. Fog. The microarchitecture of Intel and AMD CPU's: An optimization guide for assembly programmers and compiler makers , 2006

[29] Kyoung-Don Kang, Sang Hyuk Son, John A. Stankovic: Service Differentiation in Real-Time Main Memory Databases. Symposium on Object-Oriented Real-Time Distributed Computing 2002: 119-128

[30] D. Grunwald, B. Zorn, and R. Henderson. *Improving the cache locality of memory allocation*. In Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation, volume 28(6) of ACM SIGPLAN Notices, pages 177--186, Albuquerque, NM, June 1993. ACM Press.

31 James W. Grenning, "Why are You Still Using C?" Embedded.com, 2003

32 Marcel Kornacker, C. Mohan and Joseph M. Hellerstein. Concurrency and Recovery in Generalized Search Trees. *Proc. ACM SIGMOD Conf. on Management of Data*, Tucson, AZ, May 1997, 62-72.

33 M. L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo. *Supporting frequent updates in R-trees: A bottom-up approach*. In VLDB, 2003.

34 S. ˇSaltenis, C. S. Jensen, S. T. Leutenegger, andM. A. Lopez. Indexing the Positions of ContinuouslyMoving Objects. Technical Report R-99-5009, Department of Computer Science, Aalborg University (1999).