Towards Efficient Reorganisation Algorithms of Hybrid Index Structures Supporting Multimedia Search Conditions

Carsten Kropf
Institute of Information Systems, Hof University, Hof, Germany

Keywords: Database Architecture and Performance, Indexing, Spatio-Textual Indexing, Hybrid Index, Data Structures and Data Management Algorithms

Abstract: This paper presents the optimization of the reorganisation algorithms of hybrid index structures supporting multimedia search conditions. Multimedia in this case refers to, on the one hand, the support of high dimensional feature spaces and, on the other, the mix of data of multiple types. We will use an approach which may typically be found in geographic information retrieval (GIR) systems combined of two-dimensional geographical points in combination with textual data. Yet, the dimensions of the points may be arbitrarily set. Currently, most of these access methods implemented for the use in database centric application domains are validated regarding their retrieval efficiency in simulation based environments. Most of the structures and experiments only use synthetic validation in an artificial setup. Additionally, the focus of these tests is to validate the retrieval efficiency. We implemented such an indexing method in a realistic database management system and noticed an unacceptable runtime behaviour of reorganisation algorithms. Hence, a structured and iterative optimization procedure is set up to make hybrid index structures suitable for the use in real world application scenarios. The final outcome is a set of algorithms providing efficient approaches for reorganisations of access methods for hybrid data spaces.

1 INTRODUCTION

Hybrid index structures, or in general, access methods for disk oriented systems with the ability to efficiently explore hybrid data spaces have been investigated much during the last years. Basically, these access methods provide fast access to the data stored in the underlying data spaces. Many applications may use them to search in a geographic information retrieval (GIR) context utilizing a combined scheme of textual and geographical data, like vector data such as points, lines or polygons. A GIR system may use relational database systems to persist the data. Yet, specialized indexing systems are necessary supporting queries for multidimensional ranges in conjunction with keywords. Typical search conditions in this application domain are of mixed type of textual and geographical predicates. Other types of combinations, for example normalized in combination with non-normalized data sets, may also be efficiently supported by these hybrid access structures. Most of them use some kind of index structure for normalized values, e.g. a B-Tree or an R-Tree, in combination with one for non-normalized values, e.g. an inverted index or a Radix tree. Several variants for this kind of access have been created and empirically evaluated. Examples of query classes may be boolean range queries supporting checks for existence of keywords inside document texts in combinations with containment of (geographical) points inside ranges or polygons or the overlap of ranges or polygons.

Unfortunately, the validation of these access structures is mostly performed in synthetic simulation based environments. These experiments show an improved retrieval behaviour and fast access to the underlying datasets. Thus, search processes executed with these access methods perform very efficiently. In realistic environments, a GIR system might be used to store data generated by a web crawler. It could also serve for persisting data from social networks. These allow the integration of geographical data during publishing time. Either the use of standard web crawlers applying an analysis and filter chain or the integration of social media produces several millions of data portions every day. Each of these postings must be included in the hybrid index structure to allow the search and retrieval of the data based on the analysis results, again. Thus, for the integration of such a

Kropf C..
Towards Efficient Reorganisation Algorithms of Hybrid Index Structures Supporting Multimedia Search Conditions.
DOI: 10.5220/0004996302310242
In *Proceedings of 3rd International Conference on Data Management Technologies and Applications* (DATA-2014), pages 231-242
ISBN: 978-989-758-035-2
Copyright © 2014 SCITEPRESS (Science and Technology Publications, Lda.)

hybrid access method, it is inevitable to provide, besides the existence of efficient retrieval mechanisms, also fast insertion and reorganisation algorithms.

Simulation based environments which validate the enhanced retrieval capabilities are most often constructed in main memory and executed on server systems with large amounts of RAM. Sometimes, also disk oriented test suites are used to simulate the behaviour of the access methods. However, the most interesting feature validated in the experiments is the quantity of disk I/Os which may, e.g., be monitored through access counting. Generally, disk I/Os may be used as an approximation of time as still nowadays, the positioning and read/write operation of disk heads towards a rotating hard disk takes milliseconds to be completed. Thus, it is sufficient to validate the block access count using simulation based synthetic environments.

One of the main issues which lead to this work was a realistic GIR system backed by a relational database. We tried to integrate this new kind of access structures combining an R-Tree augmented with bitlists with an inverted index lookup approach. During the implementation of the structure in a realistic environment, the reorganisation time of the hybrid index structure inside the realistic database environment was one of the conspicuousnesses during performance monitoring of the web crawler which generated several thousands of postings per day. The performance of the retrieval algorithms is superior to the one of independent searches in multiple structures (inverted index and R-Tree separated from each other), but the reorganisation performance is just unacceptable for this task. Based on the blocking behaviour of table modification operations, a bad reorganisation performance also leads to high retrieval times if searches are issued during insertions, updates or deletions. Therefore, the reorganisation algorithms had to be reinvestigated in order to reduce the effort to a reasonable amount. Basically, it must be noted that not only the construction of a data structure must be investigated but also the algorithms leading to the methodology of construction and retrieval in a new kind of access structure. The reorganisation algorithms are such a gap which are not described in detail, yet. This paper describes the procedure and final outcome of an optimization based on an initial implementation of a hybrid access method which evolves to acceptable performance bounds using an iterative approach. Hence, the term “iteration” is used here to describe the single steps inside the entire optimization procedure.

The main contributions of these work are:

1. A discussion of the initial implementation as well as the definition of a test suite for the optimization of reorganisation algorithms of hybrid access structures,
2. an overview of the optimization iterations performed to limit the reorganisation effort to a reasonable amount and
3. the final state of efficient reorganisation algorithms with focus on insertions.

This work is structured as follows: Section 2 discusses in short the related work, especially hybrid index structure variants and experiments. An outline of the initial implementation as well as the definition of a test suite and the outer circumstances, like input data and parameters is given in section 3. The procedure of the actual optimization is presented in section 4 which also explains details of selected optimizations and phases. The final state of the reorganisation algorithms is given in section 5 where the final and efficient algorithms are explained.

2 RELATED WORK

Basic approaches for creating GIR systems are presented in (Göbel and de la Cruz, 2007; Kropf et al., 2011; Vaid et al., 2005). These papers introduce the requirements and also analysis techniques for GIR systems like toponym extraction and already provide first insights on possible indexing schemes. Such a system may utilize the enhanced capabilities of hybrid index structures to present the data to the user, fast.

Hybrid indexing techniques, either with focus on top-k or boolean queries are presented in (Felipe et al., 2008; Göbel et al., 2009; Göbel and Kropf, 2010; Rocha-Junior et al., 2011; Rocha-Junior and Nørvåg, 2012; Wu et al., 2012; Zhang et al., 2009; Zhou et al., 2005). Several variants of hybrid access methods are presented in these papers. The targets either include boolean queries which just ask for existence of certain features like keywords or top-k queries which already employ ranking possibilities. The main construction of these access methods is similar for most of them. One base access method is taken into account managing data from one part of the data space, e.g. the geographical/normalized data sets, and is then augmented with a representation for another part, e.g. the texts/non-normalized data sets. We use one of these methods as the base of our investigation and try to enhance the reorganisation performance.

An experimental evaluation of available structures and methods for spatial keyword processing is given as an overview in (Chen et al., 2013). Also the classes

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

of boolean range queries as well as boolean k-NN or top-k kNN queries are investigated and an evaluation for each of the problem classes is given based on pre-defined datasets. This analysis evaluates the query efficiency regarding the respective query classes. Yet, the construction and update efficiency which may be described as reorganization effort is left out of scope, there.

3 INITIAL IMPLEMENTATION AND TEST SUITE DEFINITION

This section basically outlines the initial implementation of the hybrid index structure under examination. It must be noted that a more general overview is given here and only the respective phases are discussed as well as the setup of the test suite in conjunction with test and input parameters.

The access structure under test is described from a conceptual view in (Göbel et al., 2009). The basic concept of the hybrid index implemented for this work can be seen in figure 1. It is comprised by an initial inverted index separating high and low frequently used terms based on Zipf's Law (Zipf, 1949). This empirical law is used to, on the one hand, keep the bitlists small and, on the other, allow sequential searches for elements to a pre-defined extent. In some cases, it might be more efficient to scan the results returned from an inverted index linearly instead of proceeding with the retrieval in the hybrid R-Tree part. The tuples to be stored in the index consist of a set of terms or words from textual documents in conjunction with a set of (geographical) points. The structure consists of an initial inverted index which is the starting point for all operations. All terms are indexed in this structure. For frequently occurring terms, only references to bit indices which are used in further operations are stored whereas for seldom occurring terms, all document entries, they are present in, are stored directly, there. Thus, low frequently occurring terms are directly searched for inside this inverted index. The document heap structure serves for persisting all point values from the documents currently not present inside the hybrid R-Tree to be able to support sequential searches on point data, as well. The next storage structure, which is only used for commonly occurring terms, is the hybrid R-Tree which is a basic R-Tree augmented with a bitlist. The bitlist is used to indicate the existence of a term, represented by its bit index, inside a certain subtree. As the bitlist is of fixed size, multiple hybrid R-Tree instances for discrete bit index ranges might exist. The root nodes of these R-Trees are stored inside the R-Tree Root Storage component. The final destination of each term is inside the secondary inverted index structures. Each of the leaf elements of the hybrid R-Tree which then represents a point value has one of these secondary structures assigned. The terms are finally persisted there to build the intersection of point and term assignment to document in an efficient manner. Each of the inverted index structures, initial as well as secondary, is accessed via a B-Tree which serves as the directory.

However, based on the implementation inside a realistic database environment, some changes are introduced to the original proposal. The database selected for the implementation of the access structures is a modified version of the H2 Database Engine[1]. The main changes introduced to this database system result from the requirement of adding user defined access methods to be able to load them from external places, e.g. a jar file residing in a specific folder.

The modification of the initial inverted index will be referred to as "*rootBTree*" phase. The phases which describe the modification of the hybrid R-Tree in the following work are called "*rtreeExpand*" for general R-Tree insertion operations, "*distribute*" for keyword distribution and "*generateList*" for subset generation of specific terms valid for a given point/region. The modification of the secondary indexes is referred to as "*putEntries*" in the following. As mainly the insertion is inspected, the entire reorganisation process is called "*add*".

We refer to "iteration" as one phase inside the entire optimization procedure. It must be noted that the implementation before the first optimization iteration is really a naïve one. The inverted index structures, initial and secondary, store the directory inside a B+-Tree. Each occurrence of one term and its relation to a document is marked once directly inside the leaf nodes. Hence, lots of entries inside the B+-Tree exist mapping from exactly the same term to one document they occurs in. This is obviously not the best option, but it is worked on later during the optimization iterations. After the insertion of the entries at the initial inverted index, the terms with a occurrence frequency greater than a pre-defined arbitrary user defined limit are distributed to the hybrid R-Tree. Hence, first the node elements and regions are created by inserting the points to the existing R-Tree. Then, the distribution of the terms to the inner and finally the leaf nodes of the R-Tree is executed. This is backed by the generation of a list of elements valid for the respective subtree to distribute them to subtrees where they spatially fit. The final step is the insertion at the secondary inverted index, initially also executed using the naïve

[1]http://www.h2database.com/html/main.html, accessed 2014-03-24

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

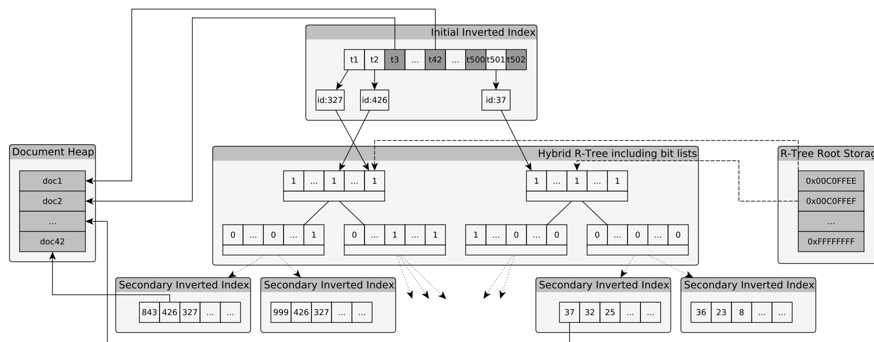

Figure 1: Conceptual Overview of the Hybrid Index Structure.

approach.

The test suite used here measures the results of specific insertion operations repeatedly. Generally, it monitors lots of features of the hybrid index structure like the number of disk I/Os (no distinctions are made between read and write) differentiated based on the page type, the highest assigned bit index and other internal states of the hybrid access structure. The information about, especially, the disk I/Os also had major influences on the decisions regarding the restructuring of the algorithms. However, primary subject of investigation for this study is the runtime of the algorithm. Hence, the test suite also contains profiling facilities used repeatedly. Based on the fact that the profiling produces large XML output files, an iterative measurement scheme is introduced which is only applied at pre-defined points in time (50 document insertions without, then one with, continuing with 50 without profiling activated, ...). By using an import to a graph database and into adopted analysis tools, an output for the respective phases may be generated displaying the total and the average runtime as well as the number of calls to each specific sub-routine.

The input data result from an analysis of a Wikipedia dump[2] which is unfortunately already deleted from the servers. Newer dumps of the Wikipedia are generated repeatedly and freely available[3]. The given analysis may be applied to newer versions of the dumps, as well. Besides, also a pre-processed version of the Reuters TRC2[4] corpus is used for the verification which is not presented in this paper. Thus, we also ensure that the given optimizations are independent of the corpus used and represent generic approaches valid for any kind of corpus. An

Table 1: Table of Settings for the Pre-tests Testsuite Run.

Parameter	Value
HLimit	200
R-Tree Elements	5
Document Count	969
Splitting Method	R-Tree (1–2)/R*-Tree (3–7)
Choose Subtree	R-Tree (1–2)/ R*-Tree (3–7)

analysis scheme is executed which uses stop word removal and porter stemming (see (Porter, 1997)). After that, coordinates are extracted from the articles, on the one hand, by using the already assigned ones which may be set up for points of interest inside Wikipedia. On the other hand, also a toponym extraction functionality is applied which detects place names and assigns discrete geographical coordinates to the place names found in the articles. The articles consisting of sets of terms and points serve as an input for the database table backed by the hybrid access method.

Unfortunately, the first iteration took very long for completing the measurement runs. Hence, only a very limited number of measurement runs could be executed. 38 repetitions of the test suite runs are performed. Basically, due to the varying approach of not monitoring 50 documents (including queries after each 25th) and monitoring one afterwards, the total amount of documents inserted to the database table is only $\lceil \frac{38}{2} \rceil \cdot 50 + \lfloor \frac{38}{2} \rfloor = 969$. During the actual research work on the optimization, the quantity of documents processed by the hybrid access method was, obviously, much bigger than the given 969 used for this analysis.

Some parameters may be set on the test suite for control. They are summarized in table 1. The parameter *HLimit* refers to the arbitrary user defined limit separating low and high frequently used terms. It is set up to 200 based on prior experiments. As a fixed block size is set for the database disk pages, either the R-Tree element count or the number of entries present to be set inside the bitlist may be specified.

[2]http://dumps.wikimedia.org/enwiki/20111007/enwiki-20111007-pages-articles.xml.bz2, accessed 2011-11-07

[3]http://dumps.wikimedia.org/enwiki/latest/, accessed 2014-05-14

[4]http://trec.nist.gov/data/reuters/reuters.html, accessed 2014-05-14

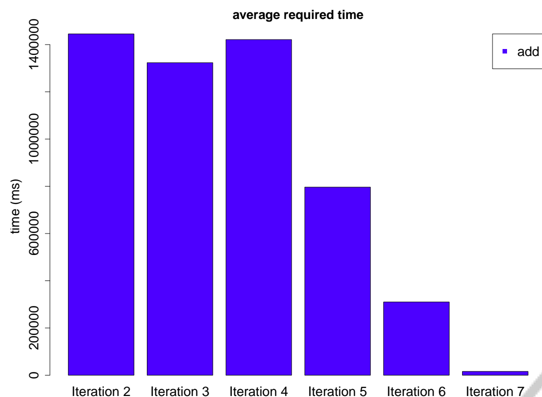

Figure 2: Overview of Average Required times for Adding an Item (Iterations 2 – 7).

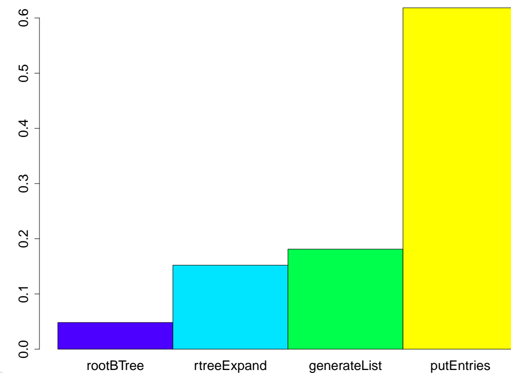

Figure 3: Average Percentual Runtime per Phase.

In this case, five R-Tree elements are configured and the size of the bitlist is calculated. The R-Tree may be adopted to the user's needs by applying different split or choose subtree methods (e.g. (Ang and Tan, 1997; Beckmann et al., 1990; Guttman, 1984)). These change between iteration 2 and 3 as result of an optimization.

As update operations in databases are often performed by marking one tuple as deleted and adding a new one, we do not especially discuss these, here. This holds (at least) for the inspected databases H2 and PostgreSQL. The query efficiency of the changed structures and algorithms is also not discussed because one main focus of the optimizations is to improve reorganisation efficiency whilst keeping the retrieval performance constant. The measured times are only compared relatively to each other. Hence, we omit a detailed description of the hardware used for the tests.

4 EVOLUTION OF HYBRID INDEX STRUCTURE REORGANISATION ALGORITHMS

An overview of the average duration of the reorganisation runs is presented in figure 2. As a side note, iteration 1 is excluded from this plot and many others following because the optimizations performed with this iteration including the cache mechanism are just too large to be reasonably displayed ($\approx 0.531\%$ relative runtime of phase "*add*" for iteration 2 in comparison to iteration 1). It must be noted that, although all following plots show average runtimes in milliseconds, the values cannot be taken as a reference. This results from the chosen profiling approach. For the time measurements, additional code for monitoring as well as exporting is executed which distorts the absolute values because the profiling approach adds a certain time overhead. Hence, they may only be used to compare the times measured during the test suite executions and may not be taken as absolute ones. Figure 2 describes the trend of iterations 2 – 7. It shows a clear decrease of the runtime for each iteration except for iteration 4. Inside the remaining phases, the differences are comparably high. Yet, a clear decreasing tendency, at least from iteration 5 to 7, can be depicted from this plot. The runtime difference of iteration 2 to 3 is only marginal whereas it slightly worsens in iteration 4. This is is a result of the size of the document set. On principle, the quantity of documents is continuously raised between the particular iterations during the real optimization iterations executed in external studies. The changes applied in iteration 4 mainly profit from a larger document set. Thus, the relatively small cardinality of the document set used in this test run does not produce the same effects as for larger sets.

The particular phases of the reorganisation algorithm are now analyzed in detail with respect to the changes performed during the optimization iterations. The values of the first iteration are omitted because, as already seen for the entire process, the differences between the first and the subsequent iterations are simply too big to be displayed reasonably.

Figure 3 shows the average percentual portion of the respective phases in relation to the entire runtime. Hence, the initial inverted index ("*rootBTree*") contributes only a small portion of the runtime whereas the remaining phases have more influence, in total. Additionally, no iteration directly refers to this structure. Therefore, this phase is left out of consideration in the following subsections.

4.1 Iterations

The iterations described in this subsection are executed in this order. That means that the sequence, presented in the following subsections, also corresponds to the one executed to reach the final goal of obtaining efficient reorganisation algorithms. The outcome of each optimization is validated based on pre- and post-tests. The facts, presented here, describe the final outcome of each of the sequentially executed optimization iterations.

As already seen in section 4, each of the optimization iterations introduces differences in the runtime. However, it is probably interesting which kinds of adoptions lead to the respective improvements. A short list of changes in the algorithms or data structures is given here. Each of the changes introduced for the respective iterations is motivated based on observations leading to solution alternatives. One of them is selected on grounds of additional evaluations (e.g. further investigations or experiments).

The iterations seen in the figures correspond with the following list of adoptions:

1. *Initial State* using the naïve implementation as described in section 3. The values for this iteration are omitted in the plots because of the tremendous differences.
2. *Page and Value Caches* introducing least recently used caching (e.g. (O'Neil et al., 1993)) for serialization and de-serialization of objects as well as an adopted caching mechanism. The used caches are introduced in addition to already present ones from the H2 database.
3. *R-Tree Distribution* which considers different methods of selecting a proper subtree or splitting a node in two. Three methods of splitting (R-Tree linear, R*-Tree and Ang and Tan) as well as two methods of subtree selection (R-Tree linear and R*-Tree) were cross-evaluated. The R*-Tree splitting and choose subtree work best for the datasets under test (Wikipedia and also an additional Reuters excerpt).
4. *Inverted Index and Storage Changes* adopting the internal storage of objects from a serialized version of the entire object to the decomposition of individual objects. This makes a direct access to stored data more efficiently. The storage mechanism of the postings inside the inverted index which is used intensively in initial and secondary index storage is changed based on experimental and theoretical considerations. Depending on the size of the posting lists compared with the available block size, they are either stored inside the directory B+-Tree or at external pages.
5. *Further Inverted Index* manipulations. This iteration includes an optimization for searches inside the directory B+-Tree. Thus, a binary search in inner elements is included reducing this internal search effort from linear $O(n)$ to logarithmic $O(\log n)$. Besides, empirical data are taken to check the occurrence frequency of all points in the datasets (Wikipedia and Reuters) and the outcome is that the frequency is also distributed based on Zipf's Law. A specialized storage mechanism is included for points occurring in exactly one document.
6. *Pre-Calculation of Item Insertions* introduces a modified version of the distribution of term to document mappings to the points where they fit in inside the hybrid R-Tree. Initially, the data structure used is set up on the terms pointing to the documents which refer to the points contained in them. This is changed to a pre-calculated hash table version of points referring to terms which themselves reference the documents. Thus, the pre-calculation is very similar to the structures required by the hybrid R-Tree and secondary inverted index, respectively. A more direct algorithm of distribution is chosen which loads leaf nodes where the points are in and the pre-calculated inverted posting sets may be inserted to the secondary inverted index structures, without further calculations. In addition, bulk-loading oriented operations for secondary inverted index structures are also introduced.
7. *Spatial Structures for Spatial Distributions* are introduced in the last of the inspected iterations. The hash table is replaced by a KD-Tree (Bentley, 1975) which is better suited for handling spatial objects. This enhances the access to spatial objects on the one hand for subset generation and enables a cache hit query in inner node elements. Thus, for inner elements inside the hybrid R-Tree, it is sufficient that at least one point exists in the set of objects to the distributed elements which is contained inside the region described by the node element to be further distributed. Therefore, no subsets must be created and spatial range queries are efficiently supported which is not given for hash tables.

4.2 R-Tree

The R-Tree operations are not investigated exclusively during the optimization. Only the distribution is changed (split and choose subtree) during iteration

SCIENCE AND TECHNOLOGY PUBLICATIONS

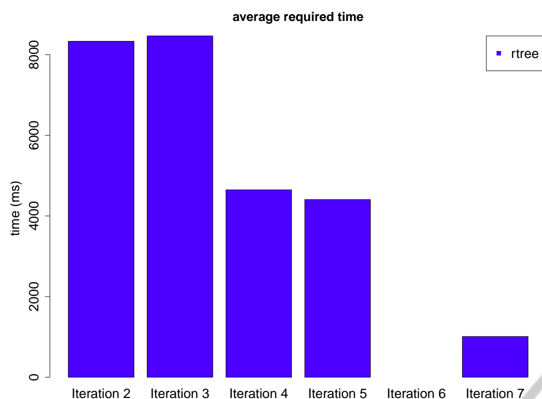

Figure 4: Overview of R-Tree Optimization Iterations.

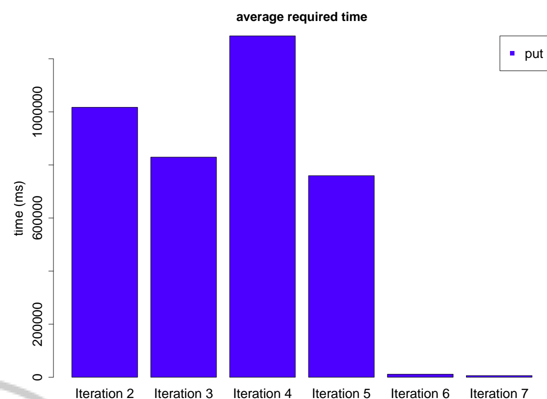

Figure 5: Overview of Secondary Inverted Index Manipulation Iterations.

3. The alterations of the runtime are mainly introduced as side effects from other optimizations. The results can be seen in figure 4. Iteration 1 is omitted here because the caching mechanism introduced in iteration 1 has too big effects on the runtime that a proper display of the results is impossible. The cache mainly works for the spatial objects stored inside the R-Tree (and some others, as well).

Although iteration 3 introduces an adopted version of the distribution, the runtime slightly rises, here. This results from the fact that the actual calculation of the improved distribution takes more time. The basic R-Tree algorithms (at least the linear version implemented here) is not very computationally complex. The more complex algorithms of the R*-Tree contribute negatively to the runtime of the insertion operations. Additionally, the change in the R-Tree element structure should support the distribution of terms to points in the "*distribute*" part of the hybrid index which follows the "*rtreeExpand*" phase, shown here. Hence, an optimization of the actual R-Tree modification is also not intended for iteration 3.

The greatest effects for the actual R-Tree manipulations are given in iteration 4 and 7. The value for iteration 6 is missing because in this iteration, another insertion and distribution algorithm is applied. It is performed together with the distribution and no phase can be isolated for analysis here. The difference between iteration 3 and 4 results from the change in the storage mechanisms, because the spatial objects may be modified more directly, then. In advance, a composed object of spatial component, bitlist and secondary inverted index reference is stored inside the elements of an R-Tree node. The new storage approach separates the storage to individual column representations. The gap present between iterations 5 and 7 is explainable because of the improvements introduced by the KD-Tree. The algorithms are substantially changed in this iteration. Up to iteration 5, the insertion and distribution of the new terms and points to be placed to the hybrid R-Tree is executed by first iterating through all points present in the set of overflowing elements, looking up the points and, if not present, inserting them. Subsequently, the distribution of terms to the points is executed. Hence, all affected nodes are accessed at least twice. The new version first distributes the terms to already present R-Tree point elements. Afterwards, all points (and affected terms) which were not handled, yet, are placed. Hence, the possibility of loading paths multiple times exists but is lower based on the more direct way of placing only the remaining elements and not having to load the entire tree (at least) twice.

4.3 Secondary Inverted Index

The effects of the optimization iterations for the secondary inverted index are presented in prior to the ones of the distribution phase because it directly influences the runtime of the distribution of elements. However, the secondary inverted index reorganisation does not depend on other algorithms and may thus be inspected exclusively.

Figure 5 shows an overview of the iterations of the secondary inverted index manipulations. Iteration 1 is omitted, again, due to huge differences of the runtime. The secondary inverted index manipulation is executed very frequently because all terms that are distributed to the hybrid index must be placed at a final destination, which is the secondary inverted index. Hence, for each point affected by the distribution of items, the manipulation of the secondary inverted index is carried out. The general tendency of this phase is continuously decreasing, too. Iteration 4, where the runtime of increases very much, is an outlier. Basically, it can be shown that for larger numbers of documents handled by the hybrid index structure, the ef-

fects introduced with iteration 4 are also beneficial for the runtime. The changes introduced in iteration 4 refer to the storage of a high quantity of documents. If there is a large number of references from one particular item stored inside the directory to a lot of entries in the document heap, a strategy is applied to the elements inside the B-Tree which, depending on the occurrence frequency of one term, decides whether to place the term inside the directory B+-Tree or in externalized posting lists. Unfortunately, the management operations introduced using this approach for the inverted index become by far more complex than the initial ones. Originally, for each element a tuple is built which is simply inserted to the secondary inverted index using the standard B+-Tree algorithms and placed directly inside the directory. After the introduction of the new strategy, decisions must be taken whether to place the element to the leaf node itself or to move it to an external page. References to these external pages must be updated if the element is moved, which includes additional computational overhead. This indicates that the algorithms for element placement in the new version of the inverted index are by far more complex than the original ones which simply took each term to document reference and placed it instantly into the B+-Tree.

The remaining tendency demonstrated there is that the average time required for one manipulation of the secondary inverted index continuously decreases from iteration 2 –7. An optimized way for the distribution of point values inside the hybrid index also seems to have a side effect for the secondary inverted index performance. Yet, the most significant changes for this manipulation phase turn out between iterations 5 – 7. The combination of a pre-calculation, an advanced inverted index and the introduction of spatial structures for the distribution algorithms seems to increase the performance of the secondary inverted index manipulation to a large extent. The effects of the reorganisation of the storage mechanism for points referencing exactly one tuple eminently affects the average runtime. The same obviously holds for the bulk-oriented insertions introduced in iteration 6. Iteration 7 actually does not change much in the basic behaviour of the algorithms and thus has nearly no influence, there. Basically, iteration 7 only introduces an optimized way for the distribution of elements throughout the hybrid index which means that, besides a pre-calculation of elements, also the distribution may be executed in a more optimal way. This change leads to the fact that the performance of the generation of valid items for a given subtree is optimized evoking, on the one hand, a lower amount of calculations of valid elements and, on the other, to

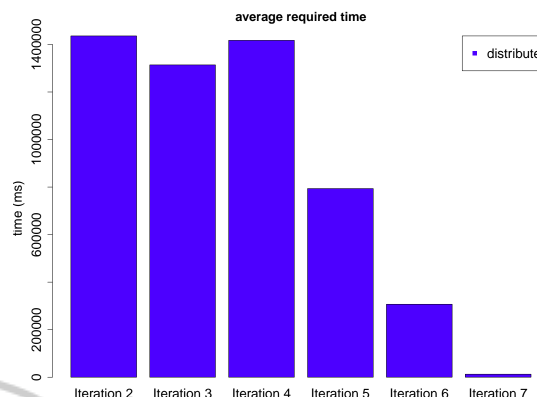

Figure 6: Overview of Distribution Optimization Iterations.

an optimized way of treating pages inside the cache of the H2 database. If a lower amount of pages is visited during the reorganisation, less pages need to be managed inside the cache for the remaining page types. This produces more space available in this cache. Hence, it is possible to perform the management of the potentially high quantity of secondary inverted index pages more efficiently because a large amount of the available cache can be used here. This leads subsequently to a more efficient utilization of the cache resource from the database. Nevertheless, the most considerable difference can be detected between iteration 5 and 6 which means that the bulk-loading as well as the appropriate storage of points occurring in only one document lead to a tremendously improved performance for the secondary inverted index structures.

4.4 Distribution of Entries

Besides the inverted index manipulation, the distribution of the entries is the phase which is worked on most considering the optimizations. It is related very closely to the secondary inverted index manipulation because the entries are placed into the secondary inverted index at the end of the distribution. Hence, the items valid for the particular subtree represented by an element are calculated and afterwards distributed to this subtree. In the case of a leaf node, the entries are subsequently inserted to a secondary inverted index. Thus, for most optimization iterations, except for iteration 6 where the distribution is omitted by including direct insertions to leaf nodes, these two phases must be inspected in connection with each other.

The results of the measurements for the distribution phase are displayed in figure 6. This figure shows that the performance for this part steadily increases as well (except for iteration 4). Iteration 1 is again omitted based on the already mentioned rea-

sons. The distribution phase contains the recursive traversal through the tree as well as the subset generation and placement of the items to the secondary inverted index structures. Thus, the efficiency of this structure directly affects the "*distribute*" phase. This influence explains the results from iteration 4. It can be seen that the alteration of split and choose subtree methods has minor effects, at least compared to the iterations carried out later on. Iteration 5, however, introduces a great performance increase. In this iteration, the storage structures are adopted, which means that points occurring only in one single document are stored without the necessity of secondary inverted index manipulations. As a side note, this is also beneficial for the retrieval effort. Hence, the process of skipping these manipulations leads to a major performance improvement here. But, the generation of the item lists is still executed there. Thus, for each R-Tree element a set of valid term ids and documents is generated to be distributed to the respective subtree, there. This is omitted in iteration 6 where the pre-calculations in the shape of a hash table are introduced. From iteration 5 to 6 a relative performance enhancement of $\approx 50\%$ can be depicted. This is a direct effect of the introduction of the hash table pre-calculation and the direct insertions to secondary inverted index structures. No subsets must be calculated during the distribution phase. The most influential part at this stage is the introduction of the hash table. Although it has proved successful for the iteration, it is its main time consumer, as well. This is primarily based on the "random" access of the map entries which has been overcome by the use of a spatial structure (KD-Tree) at the end. The effects of the KD-Tree are tremendous like the difference between iteration 6 and 7 shows. In comparison to the map, the KD-Tree shows two substantial distinctions. On the one hand, the list generation is skipped and replaced by an approach performing cache-like. Thus, the generation of potentially large lists for subtrees can be left out. On the other, the spatial structure of the KD-Tree arranges the elements better suited for queries towards specific points or ranges. This leads to an enormous performance gain between iteration 6 and 7. Besides these improved spatial distributions, the approach also benefits from the chance of not having to visit many subtrees and visiting them only once.

Figure 7 shows the results for the generation of the subsets valid for the particular subtrees. The general tendency of this figure is comparable to the one of the distribution function if the put entries phase is disregarded. Iteration 6 skips the generation of the sublists by introducing the hash table. This is changed in iteration 7 where the KD-Tree mainly functions as a comparison cache. The queries for the respective entries valid for one spatial element inside the R-Tree are performed via the KD-Tree in the last iteration. A query is only executed for presence of at least one element from the KD-Tree inside or equalling the currently inspected element. Only in the leaf level, the sets are generated which are already pre-calculated before the distribution algorithm. Thus, actually in no stage from iteration 7, a subset is generated because the already prepared ones are chosen for insertion. It is obvious (see figure 7) that this change is the most beneficial regarding the performance.

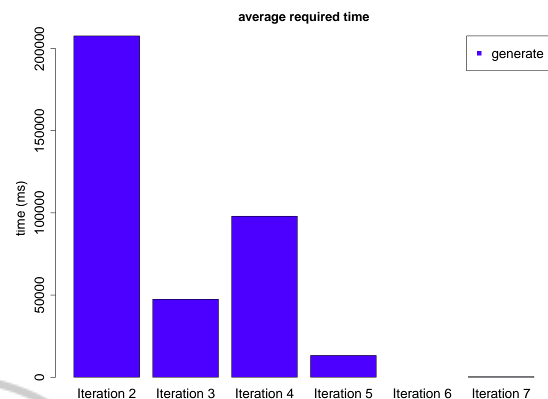

Figure 7: Overview of Sublist Generation.

With exception of iteration 4, the performance of the subset generation lowers very much during the optimization iterations. A great effect can also be depicted between iteration 2 and 3. That indicates that it profits much from the improved version of of the R-Tree distributions. Based on the adopted versions of the inverted index algorithms, the generation of subsets also benefits from the caching. The algorithm, basically, first loads all documents contained in the sets and checks the point values. Thus, if less pages are loaded by other parts, the subset generation profits from this by not requiring all the entries to be loaded from the document heap repeatedly.

5 FINAL RESULT OF THE REORGANISATION ALGORITHMS

This section provides the final outcome of the reorganisation algorithms. Basically, only the insertion is inspected in this case. The algorithms are listed in pseudo-code which should be applicable to a wide variety of programming languages. These algorithms are the final outcome of the seven implementation states as discussed in the previous sections.

Algorithm 1: Insert Document.

```
1  Function HybridIndex::Add(doc)
       Data: the document to be inserted to the
             hybrid index
2      documentHeap.Place(doc.getPoints());
3      toDistribute =
       initial.Insert(doc.getWords());
4      if toDistribute != ∅ then
5          hybrid.Insert(toDistribute);
```

Algorithm 1 shows the general hybrid index insertion algorithm. First, the document heap is manipulated. This storage structure has not been discussed, before. It is a persistence unit saving the points or normalized values for the support of sequential searches when the a term occurring inside one of the document has an occurrence frequency of lower than *HLimit*. Thus, the sequential filtering done on the documents after inverted index retrieval may be executed, there. After this insertion step, the initial inverted index modification is executed which produces a set of values to be distributed to the hybrid R-Tree which means that the term frequency is higher than *HLimit*. Thereafter, this set is further processed by the hybrid R-Tree algorithms.

Algorithm 2: Insertion at Initial Inverted Index.

```
1  Function Initial::Insert(terms)
       Data: the terms representing the
             non-normalized values
       Result: set of items to be distributed to the
               hybrid R-Tree
2      overflow = ∅;
3      for term ∈ terms do
4          inverted.FindLeaf(term);
5          if found ∧ overflow then
6              overflow.Add(element);
7          else
8              refcount =
               inverted.Insert(term);
9              if refcount > HLimit then
10                 item.SetWordId(max(termid) +
                   1);
11                 inverted.Replace(inverted
                   index item);
12                 overflow.Add(element);
13     return overflow
```

A description of the initial inverted index manipulation is given in algorithm 2. For each term in the set of terms assigned to a document, this algorithm checks whether the respective term is already in the set of terms with a high frequency. If so, they are directly added to the set to be processed, further. If not, the reference count is determined by placing the new document reference to the postings list. If after that, the term frequency is higher than the given limit, a new bit index is created to indicate the presence or absence of this term inside the bitlist of an R-Tree element, all references to this term are removed from the inverted index and it is also added to the set of terms to be further processed.

Algorithm 3: Insertion on Inverted Index.

```
1  Function Inverted::Insert(node, elem)
       Data: the node to insert the element at and
             the element itself
       Result: the frequency of the element
2      if internal storage then
3          if ! inverted.CheckSize(node, elem)
           then
4              inverted.MoveExternal(items);
5              inverted.PlaceExternal(elem);
6          else
7              inverted.PlaceToNode(elem);
8      else
9          inverted.PlaceExternal(elem);
10     return frequency
```

The general inverted index manipulation used by the initial as well as the secondary inverted index is shown in algorithm 3. Two strategies may be applied here based on the size required by the set of documents pointed to by one term. Either internal storage, which means that the references are placed directly inside the directory, or external storage indicating that an external postings page is used may be selected as strategies. In case the external storage indicator is set, the new element is directly stored, there. Otherwise, the size of the currently existing elements with the new one in addition is determined and decided whether a limit is reached. Currently, this limit is set to the size of a page, because if a second page must be created to store all references to one term in, then it is more efficient to keep the directory B+-Tree lean and place all entries to an external postings list.

The insertion of the overflowing set to the hybrid R-Tree is described in algorithm 4. First, a KD-Tree which serves for cache lookups and subset generation is constructed which is then distributed to the R-Tree root node. This KD-Tree uses the point or normalized values as keys which reference a set of terms which again refer to a set of documents each. That means that the new entries for the secondary inverted index

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Algorithm 4: Hybrid R-Tree Insertion.

```
1  Function Hybrid::Insert(overflow)
       Data: the overflowing list to be distributed
2      kdtree =
       hybrid.GenerateAssignment(overflow);
3      hybrid.Distribute(kdtree, root);
4      hybrid.PlaceRemaining(kdtree);
```

Algorithm 5: Distribution of Elements.

```
1  Function Hybrid::Distribute(kdtree, node)
       Data: the KD-Tree and the node to be
             modified
2      for entry ∈ node do
3          if ! isLeaf then
4              if kDTree.CacheHit(entry) then
5                  subtree =
                   hybrid.LoadSubtree(entry);
6                  hybrid.Distribute(kdtree,
                   subtree);
7          else
8              assignment =
               kDTree.Candidates(entry,
               kdtree);
9              secondary.PutEntries(entry,
               assignment);
10     hybrid.UpdateBitlist();
```

structures are already pre-calculated, there. Finally, new items which means that terms cooccurring with points that have not been placed, yet, are inserted to the R-Tree using the appropriate splitting and subtree choosing methods.

Algorithm 5 shows how the distribution of the elements to the respective secondary inverted index structures is executed. In the final version, for inner nodes just a check is performed whether at least one point from the KD-Tree is inside the region described by one element pointing to a particular subtree. If this is true, the algorithm recurs to this subtree until a leaf node is hit. In case of a leaf node, an assignment of term id to document is built which is valid for the respective point. This assignment consisting of potentially multiple terms which may point to multiple documents is then inserted to the secondary inverted index structures.

The last algorithm to be executed when inserting a new document instance to the hybrid index is the insertion operation on secondary inverted index structures (see algorithm 6). It checks whether currently there is only one document pointed to by the entire secondary inverted index. If this is true and the passed

Algorithm 6: Insertion Operation on Secondary Inverted Index.

```
1  Function Secondary::PutEntries(entry,
   assign)
       Data: the hybrid R-Tree entry and the
             assignment to be placed
2      if secondary.CheckSingleDocument()
       then
3          secondary.SetRowKey(document);
4      else
5          if row key not empty then
6              secondary.MoveSec(entries);
7          while assign != ∅ do
8              toAdd =
               secondary.GetEntries(assign);
9              inverted.Insert(toAdd);
```

assignment also points to solely this document, only the bitlist of the R-Tree element which references this secondary inverted index has to be updated. Otherwise, the respective assignment must be placed to the secondary inverted index by pre-calculating all elements which may be placed in a bulk-loading oriented procedure. This is executed by first searching a proper node in the directory B+-Tree and then inspecting the respective node and probably its successors if more than one element from the set of entries contained in the assignment may be placed, there.

6 CONCLUSIONS

We presented an iterative approach for the optimization of hybrid index structures supporting spatial-keyword queries. Finally, a set of algorithms is obtained which leads to efficient support for hybrid index structures in realistic database environments. An analysis of queries during the optimizations is omitted because the entire optimization work is focussed on leaving the retrieval at least constant or to improve it. Yet, the main focus was on optimizing the reorganisation behaviour. The reorganisation in this case refers to both, insertions and updates, because in many relational databases updates are performed by marking a row as invalid and inserting a new one. Hence, besides optimizing insertion operation, also the updates are improved.

The main optimization effort is done in relational database management systems. However, the final algorithms may be used in arbitrary environments where disk oriented persistence and buffer management and allocation is an issue. Hence, they may also

be ported to other kinds of persistence systems like graph databases.

A data structure always consists of the basic storage structure definition in connection with a set of algorithms to handle the data flow inside this structure. The data structure in connection with retrieval algorithms was already present in advance of this work. Yet, the reorganisation algorithms were missing. This gap is now closed as we presented efficient algorithms for this task.

REFERENCES

Ang, C.-H. and Tan, T. C. (1997). New linear node splitting algorithm for r-trees. In *SSD '97: Proceedings of the 5th International Symposium on Advances in Spatial Databases*, pages 339–349, London, UK. Springer-Verlag.

Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. (1990). The r*-tree: An efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19(2):322–331.

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517.

Chen, L., Cong, G., Jensen, C. S., and Wu, D. (2013). Spatial keyword query processing: an experimental evaluation. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 217–228. VLDB Endowment.

Felipe, I. D., Hristidis, V., and Rishe, N. (2008). Keyword search on spatial databases. *International Conference on Data Engineering*, 0:656–665.

Guttman, A. (1984). R-trees. a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, New York, NY, USA. ACM.

Göbel, R. and de la Cruz, A. (2007). Computer science challenges for retrieving security related information from the internet. *Global Monitoring for Security and Stability (GMOSS)*, -:90 – 101.

Göbel, R., Henrich, A., Niemann, R., and Blank, D. (2009). A hybrid index structure for geo-textual searches. In *Proceeding of the 18th ACM conference on Information and knowledge management*, CIKM '09, pages 1625–1628, New York, NY, USA. ACM.

Göbel, R. and Kropf, C. (2010). Towards hybrid index structures for multi-media search criteria. In *DMS*, pages 143–148. Knowledge Systems Institute.

Kropf, C., Ahmmed, S., Göbel, R., and Niemann, R. (2011). A geo-textual search engine approach assisting disaster recovery, crisis management and early warning systems. In *Geo-information for Disaster management (Gi4DM)*.

O'Neil, E. J., O'Neil, P. E., and Weikum, G. (1993). The lru-k page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 297–306, New York, NY, USA. ACM.

Porter, M. F. (1997). Readings in information retrieval. chapter An algorithm for suffix stripping, pages 313–316. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Rocha-Junior, J. a. B. and Nørvåg, K. (2012). Top-k spatial keyword queries on road networks. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 168–179, New York, NY, USA. ACM.

Rocha-Junior, J. B., Gkorgkas, O., Jonassen, S., and Nørvåg, K. (2011). Efficient processing of top-k spatial keyword queries. In *Proceedings of the International Symposium on Spatial and Temporal Databases (SSTD)*, volume 6849 of *LNCS*, pages 205–222. Springer.

Vaid, S., Jones, C. B., Joho, H., and Sanderson, M. (2005). Spatio-textual indexing for geographical search on the web. In *9th International Symposium on Spatial and Temporal Databases SSTD 2005*, volume 3633 of *Lecture Notes in Computer Science*, pages 218–235.

Wu, D., Yiu, M. L., Cong, G., and Jensen, C. S. (2012). Joint top-k spatial keyword query processing. *Knowledge and Data Engineering, IEEE Transactions on*, 24(10):1889 –1903.

Zhang, D., Chee, Y. M., Mondal, A., Tung, A. K. H., and Kitsuregawa, M. (2009). Keyword search in spatial databases: Towards searching by document. *Data Engineering, International Conference on*, 0:688–699.

Zhou, Y., Xie, X., Wang, C., Gong, Y., and Ma, W.-Y. (2005). Hybrid index structures for location-based web search. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 155–162, New York, NY, USA. ACM.

Zipf, G. K. (1949). *Human Behaviour and the Principle of Least Effort: an Introduction to Human Ecology*. Addison-Wesley.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS