Efficient Reorganisation of Hybrid Index Structures Supporting Multimedia Search Criteria

Vom Institutsrat des Internationalen Hochschulinstituts Zittau
genehmigte

DISSERTATION

zur Erlangung des akademischen Grades

doctor rerum politicarum

Dr. rer. pol.

vorgelegt

von Dipl.-Inf. (FH) Carsten Kropf, M.Eng.

geboren am 19.01.1985 in Selb

Gutachter: Prof. Dr. rer. pol. habil. Thorsten Claus, Zittau
Prof. Dr. rer. nat. Richard Göbel, Hof
Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill, Dresden

Tag der Verteidigung 21.11.2016

ABSTRACT

This thesis describes the development and setup of hybrid index structures. They are access methods for retrieval techniques in hybrid data spaces which are formed by one or more relational or normalised columns in conjunction with one non-relational or non-normalised column. Examples for these hybrid data spaces are, among others, textual data combined with geographical ones or data from enterprise content management systems. However, all non-relational data types may be stored as well as image feature vectors or comparable types. Hybrid index structures are known to function efficiently regarding retrieval operations. Unfortunately, little information is available about reorganisation operations which insert or update the row tuples. The fundamental research is mainly executed in simulation based environments. This work is written ensuing from a previous thesis that implements hybrid access structures in realistic database surroundings. During this implementation it has become obvious that retrieval works efficiently. Yet, the restructuring approaches require too much effort to be set up, e.g., in web search engine environments where several thousands of documents are inserted or modified every day. These search engines rely on relational database systems as storage backends. Hence, the setup of these access methods for hybrid data spaces is required in real world database management systems.
This thesis tries to apply a systematic approach for the optimisation of the rearrangement algorithms inside realistic scenarios. Thus, a measurement and evaluation scheme is created which is repeatedly deployed to an evolving state and a model of hybrid index structures in order to optimise the regrouping algorithms to make a setup of hybrid index structures in real world information systems possible. Thus, a set of input corpora is selected which is applied to the test suite as well as an evaluation scheme.
To sum up, it can be said that this thesis describes input sets, a test suite including an evaluation scheme as well as optimisation iterations on reorganisation algorithms reflecting a theoretical model framework to provide efficient reorganisations of hybrid index structures supporting multimedia search criteria.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

LIST OF ALGORITHMS

1 INTRODUCTION

1.1 PROBLEM STATEMENT

During the last years, great changes in information technology have been introduced. New web search technologies facilitate the quick access to information. In the beginning, the retrieval techniques on-hand were not very sufficient. Users had to know where to gather information and did not have any search technologies available for assisting them to find it. Nowadays, search engines like Google[1] or Bing[2] implement methods for making information sources available to users. Full text extraction including boilerplate removal and further analysis techniques are obtainable which derive structured information from unstructured texts as well. Hence, also dates or names of organisations or persons may be augmented to the actual search results to assist users during their searches.
Besides this structured information sophisticated methods for efficient storage and retrieval had to be developed, too. Inverted index technologies which allow the retrieval of texts via keywords facilitate the search for textual documents. Additionally, positional information may be included in this indexing technology for the support of phrase searches. Result ranking is another important part of these web search engines or information systems. It allows to order the search results based on predefined metrics to present the most relevant outcomes to the user, first.
Textual retrieval, however, is not the only application domain of information retrieval. A lot of other domains are present which also rely on structured information. Classical normalised data sets are very important in this application domain, too. Especially regarding applications in enterprises, different challenges exist. Enterprise resource planning systems like SAP[3] or Microsoft's Dynamics[4] invoke challenges for the efficient retrieval of, e.g., contract data or other kinds of structured information. Besides these enterprise related applications, also geographic data have gained increasing interest over the last years. The extensive use of smartphones contributes to this development of the importance of geographic data. Smartphones, generally, include technologies for making positional information available and use geographic data, e.g. for routing or finding nearby information as well. Therefore, the domain of geographic information systems or the mix of geographic with textual information (geographic information retrieval (GIR)) becomes progressively interesting, too.
As to the retrieval, GIR data have been investigated during the last years. Search engines supporting this task have been developed and successfully established. Some main challenges for this type of search engines are, on the one hand, the extraction of structured information

[1] http://www.google.de, accessed 2014-02-28
[2] http://www.bing.com, accessed 2014-02-28
[3] http://www.sap.com/, accessed 2014-02-28
[4] http://www.microsoft.com/en-us/dynamics/default.aspx, accessed 2014-02-28

from unstructured web documents and, on the other, the efficient storage and retrieval of these data types in hybrid data spaces. New data management techniques for this kind of challenges have also been generated.
A group of access structures, called hybrid index structures, with the ability to manage data of heterogeneous types, has been introduced to address these challenges. Many different kinds of hybrid index structures, mainly with the target to answer GIR related queries, have been created. Examples for these structures are the bR*-Tree [104] or the IR^2-Tree [30] which combine a spatial structure, like the R-Tree [36], with an annotation of textual occurrences present below a certain node. Yet, also other variants of these access structures exist which use, e.g. inverted files [105] instead of bitmaps or signature files. Another variant is to exchange the spatial structure as well and to switch, e.g., to grid based [98] variants. Although some of the proposed alternatives are quite similar to each other, they may also differ in the way and the kind of queries to be answered. Different types of queries may be fulfilled like a multidimensional range in conjunction with a set of keywords ranked or unranked as well as top-k queries which allow the selection of the *k* most relevant elements. The target of the queries has already major influences on the construction of the access method. This already shows that not all access structures are feasible for each task.
In order to respond to typical queries in geographic information retrieval oriented domains like the SARA2 search engine [38, 64] or also the combination of a document management and enterprise resource planning system, the query class of boolean range queries is the most interesting. It checks the presence of keywords, e.g. inside the document management system, in conjunction with a query range (e.g. prices or dates) in the enterprise resource planning system. Typical queries in GIR consist of a set of textual keywords and a geographical, mainly two-dimensional, range with the restriction that all results need to be included in the range and must also comprise the textual occurrences.
Typically, the introduction of new data structures and access methods contains the construction of the components, e.g. storage structures, with a set of algorithms. Unfortunately, present approaches and articles solely care about the basic construction of the indexing method and mainly illuminate the algorithms from a conceptual point of view. Additionally, they are evaluated in a simulation oriented synthetic main memory based environment. Realistic systems include database systems because the quantity of data portions to be stored is too large to be managed in main memory. Each of the structures is targeted towards the use on secondary memory like hard disks in database management systems. Yet, the evaluation of these methods mainly refers to node counting approaches with the ability to approximate the time effort and to provide an estimation of the complexity.
In order to make the setup of hybrid index structures in database management systems possible, it is inevitable to implement them in a more realistic scenario. Therefore, an implementation inside the PostgreSQL[5] database was done. It turned out that the retrieval effort of hybrid access structures in realistic databases is clearly superior to the one of a combination of classical access structures [40]. Yet, this is not true for the reorganisation effort. Due to the more complex construction of the hybrid index structures, also the distribution of the values to the proper storage elements is more demanding. This may lead to unacceptable time constraints in realistic systems.
Unfortunately, the basic description of the algorithms is not focused on disk based environments. In order to overcome the weaknesses in the current definitions, this thesis utilises a currently present hybrid access method [39] to provide algorithms for the efficient reorganisation of hybrid index structures supporting multimedia search criteria. In addition, a theoretical description should be derived to model the behaviour and properties of hybrid index structures.

1.2 MAIN PURPOSE OF THE RESEARCH WORK

This section outlines the main purpose of this research work. An introduction to the problematic key features which have led to this research work is outlined. Besides the introduction of the original problems, the research questions as well as the research method are described which form the basis of the research work to be executed.

[5] `http://www.postgresql.org/`, accessed 2014-08-27

1.2.1 Research Questions

Currently, more and more research is done towards geographic information retrieval systems, e.g. [38, 56, 64]. These systems support the search for combined geographical and textual data and are generally represented in the form of an interactive retrieval system. The approaches developed in these systems do not only apply to geographic information retrieval. The general form of a search query in this case contains a set of non-normalised attributes and a range or multiple single values of normalised attributes. Examples for non-normalised attributes may be found in full text searches. In these search approaches, a text composed by many singular terms is decomposed and stored inside a database index like the inverted index. Normalised attributes can be found inside conventional relational database systems like string or numeric attributes. Creating ranges or sets of normalised attributes does not only appear in geographic information retrieval systems but also in enterprise content management. In the latter, a enterprise-wide search solution could query for textual data, for example from a document management system, in conjunction with numeric data, e.g. from an enterprise resource planning system. In both cases, enterprise-wide search solutions as well as geographic information retrieval, it is often necessary to combine these searches, e.g. when selecting a geographical range in connection with keywords or when selecting a time range in conjunction with a query string. This shows the connection or general applicability of the approaches.
Current research is mainly executed in simulation based environments. There, implementations of structures solving the issues of searches in hybrid data spaces are performed in main memory or on the hard disk without the utilisation of a database. This is a valid approach for determining the parameters and circumstances for a successful setup of these structures. The experiments then validate a theoretical and practical deployability in realistic applications. Yet, they cannot predict the behaviour of the access methods in real database systems. Some features given in authentic applications are either hard to simulate, cannot be dissembled or are simply not the subject of investigation of the studies and are therefore left out of consideration. Nevertheless, the applicability of these structures is a crucial task for the real utilisation of new approaches. Hence, it is inevitable to take a less synthetic setting into account, too, and implement the available solutions in more realistic scenarios. Existing database systems should be extended to support the new storage mechanisms in evaluating the capabilities there. For this reason, it is expected that new or other drawbacks arise based on the totally different aim and construction of realistic systems compared with synthetic simulations.
As a preparation step, a hybrid index structure was developed inside a realistic database environment. The basic implementation is done within the PostgreSQL[6] database management system [40]. During the initial implementation, issues regarding the reorganisation performance of these hybrid index structures within realistic database scenarios arose. Inside a typical laboratory environment in a simulation based application, the main problems did not exist as a result of the differences between the theoretical framework set up in a simulation in contrast to a realistic database environment. In realistic environments, there are additional constraints because of the work executed directly on a hard disk whereas simulation based approaches mainly work in main memory. Some other database related properties are also ignored in the initial implementation as the structures are primarily constructed theoretically and validated in a simulation framework. However, the final target of an index structure is its capability to run in realistic scenarios. Therefore, a way to include theoretically efficient index structures in realistic scenarios should be found.
It could be shown that searches inside these structures work more efficiently than in a conjunction of other cases. Yet, reorganisation is still a crucial task as it also blocks concurrent search queries on the data. As a goal for a typical information system is to deliver data to the user fast, it is necessary to introduce new models of hybrid access structures in order to increase the reorganisation performance and make its inclusion in realistic database oriented scenarios possible.
These basic causes led to this research work. The main questions are how to optimise and validate such a structure. Additionally an evolution of hybrid index structures regarding reorganisation performance so that they run efficiently in realistic information system environments must be present.
Consequently, the following questions must be treated:

[6] `http://www.postgresql.org/`, accessed 2013-10-18

1. Which parts of the algorithms inside the reorganisation process of hybrid index structures lead to the currently present negative performance measures?

2. Is it possible to enhance the performance of the reorganisation algorithms as measured in advance so that a sufficiently efficient performance can be achieved whilst not affecting the search time complexity negatively by optimising first the model of a hybrid index structure and afterwards implement the model evolutions iteratively?

3. Is there a general model for hybrid index structures supporting simultaneous searches in hybrid data spaces for heterogeneous data types which may be used in multiple application domains?

In the ensuing subsections some preliminary considerations are given to each of the preexisting research questions, which lead to the construction of the entire aim of the work.

1.2.1.1 Identification of Performance Issues

The first research question (see item 1) relates to the identification of bottlenecks inside the reorganisation process of hybrid index structures. This, basically, refers most to the insertion performance. Deletion or updates do not have a similarly significant influence here. Largely, in databases or information systems data are either added or searched for. Therefore, the most important property while identifying performance issues during reorganisations is the identification of bad performing parts of algorithms inside the insertion process.
Basically, a framework must exist which can measure the efficiency of a certain model of a hybrid index structure in a prototypical development environment. For this reason, in order to derive the bottlenecks, besides a basic implementation in a well defined database environment, a test suite must be specified fulfilling the task of measuring specific parts and properties of the program. On principle, as performance tuning is one of the main goals of the research work, a systematic time measurement approach must be present on a pre-defined set of entries. That means that at the beginning of the research work, a well defined tool set as well as input values and basic implementation are required.
First, a database system to implement the structures in must to be chosen and probably be extended towards the support of loading customised index access methods. As a next step, the hybrid index structures, including the basic structures which compose the hybrid index, must be implemented inside this realistic database environment. Subsequently, it is necessary to define the measurement process in order to get replicable measurements which may be compared per iteration. This includes tests and probably training data sets as well as the measurement itself. Consequently, the parameters which will be measured, e.g. the number of loaded blocks within the database system, must be defined to derive standardised measurements which may be combined to a representative set of values. Besides the actual measurement, an analysis system must be defined which evaluates the data and has the ability to compare them (either iteration-wise or globally) in order to document the development process and model evolution of the hybrid index structures. The exact sequence of pre- and post-tests based on one iteration approach needs to be defined, additionally.
These actions briefly describe the approach how to solve the first research question. Further details are given in the respective chapters.

1.2.1.2 Optimizability of Hybrid Index Structures using a Model Evolution

The second major research question (see item 2) refers to the optimizability of the currently existing hybrid index structures as well as the evolution of the general basic model of a hybrid access structure. The first basic implementation of the hybrid index structure is only a provisional prototype which shows significant performance issues. Therefore, it must be optimised with respect to reorganisation performance as obvious issues arise in these algorithms. The retrieval algorithms work sufficiently well regarding the given reorganisation performance. Yet, also these retrieval methods need to be revalidated in order to check for proper work. Based on the measurement scheme which must be developed during the work on the identification of performance issues (see subsection 1.2.1.1), they must be detected to be optimised in the sequel.
To solve these issues, an evaluation scheme must be developed which shall be applied in each stage of the research work. It bases on the results of a test suite which presents values derived from measurements using realistic data sets. Because of these measurements, inefficiencies are displayed that must be validated by inspecting the model of the hybrid index structure. Due to the model information, theoretical literature needs to be appraised in order to obtain optimisation capabilities for the currently inspected inefficiencies. Besides the actual model inspection, the source code of the algorithms requires to be investigated. These analyses then lead to several optimisation alternatives which must be evaluated before selecting one definite way to optimise a certain program part. The final decision is taken on grounds of further theoretical evaluations and empirical measurements in order to generate a meaningful decision support. Thereafter, the model is adjusted and the changes are implemented in the source code part. The result is then target of investigation for further optimisation runs in subsequent iterations. Most probably, an iterative process will be chosen which takes only smaller changes into account leading to a higher number of optimisation iterations. This process is more safe than applying bigger changes, because big changes can be decomposed to smaller change sets and potential errors can thus be avoided in earlier states.

1.2.1.3 Generalisation of Hybrid Index Structure Models

The next question (see item 3) refers to a generalisation of hybrid index structure models. Currently, the hybrid index structures applied here mainly refer to the use in geographic information retrieval contexts. However, it can easily be shown that with small modifications, also general set represented values in combination with standard relational data types can be supported efficiently. This already covers a wide range of applications deploying this exact setup of the hybrid index structure and shows that the currently present hybrid index structure may also support other data types than geographical coordinates in conjunction with textual instances. Yet, there is no general way of expressing the capabilities of hybrid index structures. That means that the question if it is possible to outline the general properties of such access methods and how to set them up properly in order to follow a generalised model approach exists.

1.2.2 Research Method

Here, the chosen research method is delineated in short. It is closely related to the actually selected process model for this work.
In general, this thesis describes the development and improvement of database access methods. They can be thought of data structures used to efficiently manage certain specific data types internally and allow insertion, deletion, updates and search request via a well-defined interface. As the main work here is performed on a relational database using SQL as a query language, the interface to be used is exactly this language supporting `INSERT`, `UPDATE`,

`DELETE FROM` and `SELECT` statements for data definition and control. However, the approaches developed here, do not require such an interface and may also be used in general purpose setups.
The development of data structures can be thought of as a "creative process" which is conducted to develop alternative data structures and algorithms on them thus overcoming the efficiency problems of existing hybrid index structures used as a basis for this thesis. This strategy is embedded in a systematic research approach supporting the analysis and evaluation of different options. The research method deployed here to develop new or changed data structures and algorithms applied to the hybrid data space retrieval problem with special respect to reorganisation algorithms is described now.
There are, at least, two major research paradigms in information systems, currently. On the one hand, there is the descriptive view which roots in the natural sciences [76]. This research paradigm primarily works in a descriptive manner which observes the behaviour of objects. Objects, in this case, might be considered as, e.g., the observation of the influences of subjects towards their environment or vice versa. This paradigm is used in information systems research to study the behaviour of individuals regarding an IT artifact. Basically, it mainly inspects the behaviour and usage of IT artifacts. Besides this, natural science approaches tend to include research activities of discovery and justification or setting up theories to be justified, afterwards. On the other hand, there is also a second paradigm which is focused towards the creation of artificial objects. In contrast to natural sciences, it may be seen as prescriptive. Simon [93, page 114] describes design oriented sciences as approaches for "devising artifacts to attain goals". This leads to the creation or invention of so-called "artifacts" serving a special purpose in a pre-defined manner. The outputs of design oriented research include constructs, models, methods and implementation [76]. In this sense, constructs define the vocabulary of the domain which are used on conceptualisations for the definition of terms applied for the description of tasks. Models can be seen as a basis for the expression of relationships among constructs [76] whereas methods can be employed to derive a pre-defined sequence of steps to perform a task. Thus, a central task is to deduce methodological tools which can be used by natural scientists. Finally, the implementation or instantiation is a prototypic realisation of an artifact in a well-defined environment. Besides these preliminary introductions of vocabulary, also two central tasks of building and evaluating an artifact are defined in [76].
This is merely a generic distinction between natural and design science. Work is also performed to detail the actions and environment for the specific task of design science in information systems research. Generally, Hevner et al. [49] state that studying the nature and behaviour of an artifact and the design of it are two sides of the same coin.

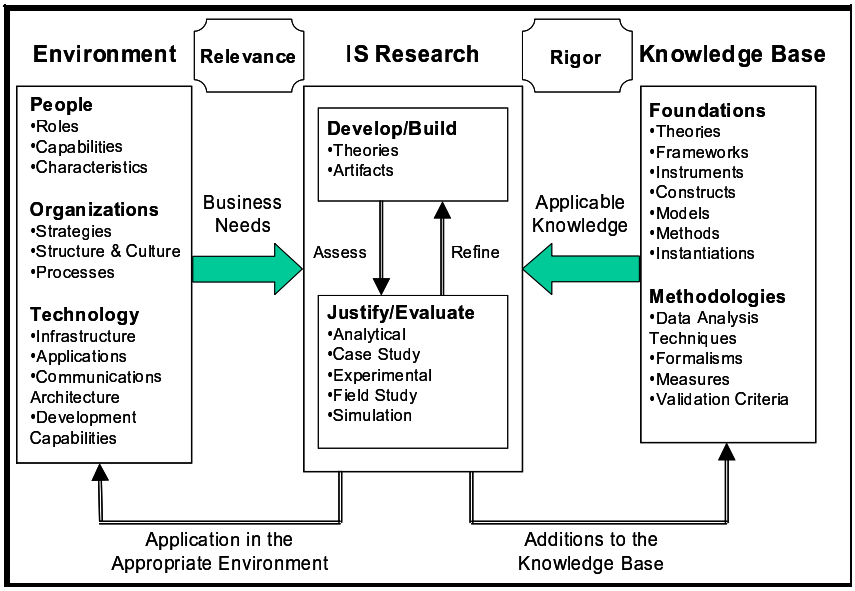


Figure 1.1: Conceptual Framework for Information Systems Research (adopted from [49])

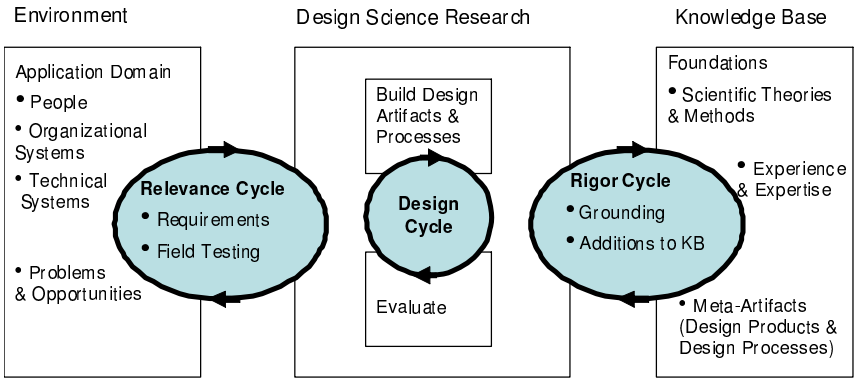


Figure 1.2: Extended Framework for Information Systems Research (adopted from [48])

Hevner et al. propose two frameworks to embed information systems research in [49, 48]. The original framework shown in figure 1.1 demonstrates three participants of information systems research: environment, information systems research and knowledge base. Environment defines the problem space [93] and thus imposes the need for the execution of the research. In the case of the present work, the original search engine approach [64] introduced challenges for indexing. Knowledge base, however, proposes the foundations and methodologies used in the application domain. The central part is then the development and evaluation part which applies methods and already present solutions from the knowledge base to derive artifacts serving the business need. The refined framework, which can be seen in figure 1.2, also includes cycles between each of the parts whereas there is a central design cycle of building and evaluating artifacts iteratively until a sufficient improvement can be seen. The steps need a rigorous application of well-defined methodologies in conjunction with the application of creativity and trial-and-error searches [49]. In addition, seven guidelines are derived for successful research in information systems which are discussed in subsection 1.2.2.1. The trial-and-error as well as the creativity based combination of tasks is found in this work repeatedly in chapter 5. The three cycles introduced in [48] also refer to creating a prototypic IT artifact which may then be used by practioners to build final solutions. Moreover, natural scientists may theorise and justify the results from design science research. Hence, design and natural sciences are also inseparable and must be inspected together.
Since the introduction of design science for information systems research, the framework has continuously been refined and analysed including ontologies and the derivation of, e.g., twelve additional theses to follow when performing design science research in information systems [51]. Further extensions also address internals of the cycles, like [100], decomposing the problems into practical and knowledge based ones. Besides, the regulative cycle is taken into account, too. This cyclic approach for the creation of solutions is a more detailed expansion of the design cycle and also applied in this thesis. It is described in more detail in subsection 1.2.2.2.
The connection between the natural or behavioural and the design science based approaches can be depicted from figure 1.3. It shows that the outcomes (IT artifacts) from design science are generally studied by behavioural science approaches deriving theories and justifications leading to truth. This truth about the behaviour of the IT artifacts is again taken into account in design science to adopt the artifacts for the presumed needs.
Based on the research questions presented in subsection 1.2.1, the goal of this work is to conceptualise, find a way to analyse, evaluate and improve hybrid access structures, especially the reorganisation and manipulation algorithms. Due to the distinction made in this subsection, it is obvious that the observation of users or organisations towards the behaviour of hybrid access methods is not the primary target in this research work. All tasks including the application of creative and innovative approaches for algorithm development may be solved by employing methods from the design science oriented point of view. The model as well as the optimised algorithms are realised in a prototypic artifact inside a realistic database environment. Therefore, the final results of this work may directly be ported to any other relational database management system to be applied in real-world information systems. Hence, the methodology deployed in this thesis is design science oriented. The following subsections additionally detail the application of design science methodologies in this thesis.

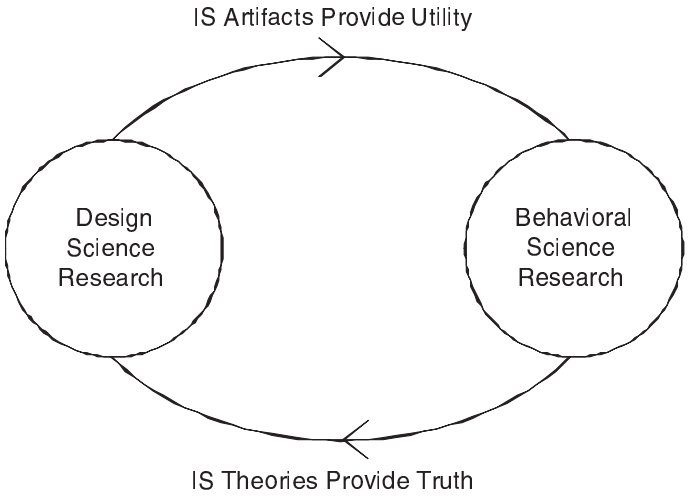


Figure 1.3: Connection Between Behavioural / Natural and Design Science (adopted from [47])

1.2.2.1 Design Science in Information Systems Research

The basic research method for this work is grounded in Design Science in Information Systems Research [49]. As this research work is mainly performed in an information systems setting and its target is to develop a working prototype of a solution using an optimisation based approach, this research paradigm may be applied here. The main deliverable of a research work set up in a design science oriented approach is an IT artifact which can be applied by users later on. Currently, it can only be stated that searches work efficiently. Yet, effort needs to be taken in order to build models of improvements which lead to a real life applicability of these new accessing methods. Therefore, the solution in the form of an IT artifact which builds the final outcome of the work is a theoretically and practically investigated hybrid index structure which may be set up in practically relevant information systems. These information systems could then support enterprise-wide search approaches in order to leverage the gathering of data for decision makers in enterprises. This indicates that one important goal is to define theoretically and practically valid solutions from a theoretical framework of hybrid index structures which will be helpful for support systems in future solutions.
Hevner et al. [49] define seven guidelines for the application of design science in information systems research. These guidelines are investigated in the following and applied to the problem stated for this research work.
The seven guidelines defined for the work in a design science oriented environment in information systems research can be seen in figure 1.4 in an overview. They will now be discussed in order to find out how to embed them in the current research work.

1.2.2.1.1 Design as an Artifact The first guideline is to produce an artifact as the final outcome of the research work. That indicates that some kind of deliverable should be created. In this case, it is not exactly defined what the concrete outcome should look like. It may be present in various forms, such as construct, model, method or instantiation. As in this research work, on the one hand, a model of generalised hybrid index structures is planned and, on the other, a method for the optimisation of an index access structure and its construction is designed, the definition for the first guideline may easily be applied to this research work. Furthermore, this work contains descriptions of concrete application fields which may benefit from the support of the hybrid index structure created here or from others which can be derived from the general model description. The results of this work can be found in chapters 4 and 5. In addition, section 6.1 shows the final outcome of the reorganisation algorithms in a pseudo-code variant.

Guideline	Description
Guideline 1: Design as an Artifact	Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.
Guideline 2: Problem Relevance	The objective of design-science research is to develop technology-based solutions to important and relevant business problems.
Guideline 3: Design Evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.
Guideline 4: Research Contributions	Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.
Guideline 5: Research Rigor	Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.
Guideline 6: Design as a Search Process	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.
Guideline 7: Communication of Research	Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.

Figure 1.4: The Seven Guidelines for Design Science in Information Systems Research from [49, page 82]

1.2.2.1.2 Problem Relevance As a second guideline, the problem relevance is given. Today, an integrated search method over all data contained in information systems is often required in enterprises . Therefore, enterprise content management systems integrate a meta-search on top of the respective data, e.g. from document management systems or enterprise resource planning systems. In order to integrate data and make them explorable in an efficient manner, it is necessary to provide proper searching support. Hence, indexing methods need to be provided which allow a quick searching of the respective data sets. As these systems deal with a quite heterogeneous data base, also adapted methods must be provided to explore the high dimensional feature space. Besides the setup in geographic information retrieval systems, the hybrid index structures which are taken as a basis for optimisation in this work may serve by doing exactly this task and integrate the heterogeneous data types for simultaneous search strategies. So this shows that, besides the geographic information sector, also the business sector is affected by the development of efficient and highly integrated search solutions.

1.2.2.1.3 Design Evaluation The design evaluation postulated by the seven guidelines may easily be applied in this case. It is obvious that the methods constructed in this work should result in some performance optimisation of algorithms and software systems. In order to verify the actual improvements of the outcome of a new iteration it is necessary to measure the algorithms repeatedly in the same setup as in advance. Therefore, a post-evaluation in each step as well as a final validation regarding the entire runtime constraints must be executed to show the validity of the given performance optimisation approaches. During the actual work, it is inevitable to inspect the performance measurements as well as evaluations for figuring out the next steps in the optimisation procedure thoroughly. Besides the actual measurements, it is also possible to evaluate the resulting structures within a realistic database scenario which may be set up inside a geographic information retrieval system. This is actually an obvious and properly quite easy way of demonstrating the optimised performance as the implementation will be done in a realistic database and therefore can directly be included inside some prototype applications which can be tested intensively in real life production environments. This indicates that the design evaluation will be performed, on the one hand, by instantiating the index structures in a real life system by further measurements carried out during the optimisation procedure and additional index structures can be created from the initial structure for other application domains, too, as one goal of the research work is to create a generalised model of the hybrid index structures. In this case, mainly an analytical, experimental as well as testing based evaluation will be executed. The evaluation of the design occurs in each optimisation iteration. First, design

alternatives are proposed, one of which is selected as the final implementation at the current stage, which is subsequently evaluated regarding the relative improvement of the respective phase. This is done repeatedly in chapter 5. Finally, an additional measurement is executed in section 6.4 which displays the overall reorganisation process.

1.2.2.1.4 Research Contributions The next guideline refers to the actual research contributions as an outcome of the research work. That means highlighting the novelty and interesting passages inside the research work. Here, the basic search for a generalised model as well as the development of a practically applicable hybrid index structure may serve as an example. At present, the hybrid index structure is not suitable for "daily use" as the reorganisation complexity is too high and some algorithms tend to be inefficient. By combining currently present research approaches with the new algorithmic of hybrid index structures, new algorithms as well as approaches are created which result in practically realisable new methods of index access structures. Besides these access structures, also a model evolution will take place which evolves from a very basic model of algorithms to more and more precise definitions of placement (or deletion) strategies in hybrid index structures. So, mainly two aspects, novelty and generality, are affected by this thesis by developing new methods of the reorganisation procedure which lead to a better understanding of hybrid index structures and their applicability and limitations. In total, the publication of the model based approaches (see chapter 4) as well as the final results 6.4 and the algorithms 6.1 are the central target of the contributions. The results are then contributed to the knowledge base and may be applied to real-world problems or be set up in information systems.

1.2.2.1.5 Research Rigour Scientific rigour is the fifth basic guideline for design science in information system research. This thesis tries to achieve it by trying to identify empirical data sets often used in information systems targeted by the access methods. Besides the standard data sets for information retrieval related constructions, also well-known measurement techniques, like profiling, or mathematical descriptions of modelling approaches for the evolving versions of the access structures are employed in order to create replicability of research methods. Thus, a well-defined measurement scheme using dynamic code analysis by applying aspect oriented programming as well as a properly defined analysis scheme is provided (see section 5.1.4). This measurement and analysis approach is defined in the beginning and utilised repeatedly to measure the progress of the research work.

1.2.2.1.6 Design as a Search Process The search process used in design science can be done in multiple ways. On the one hand, an idea to conceive an information system which is then planned and implemented might exist. On the other, there are also ways of creating solutions in the domain of information systems using heuristics or test driven methods. So, tests and evaluations obviously lead to parameter measurements which then refer to the design of alternatives subsequently tested and reintegrated in the design phase. This approach is used here iteratively by testing and implementing a software product which is refined based on the outcome of test procedures and model evolutions. Therefore, a cycle is created that applies test data as an input to generate a new version of a hybrid index structure which is tested for further improvement possibilities afterwards. The search process in this case recurs in this cycle as alternatives must be evaluated on the basis of a particular problem arising. Then, the final decision is to choose one of the alternatives in the end and revalidate the entire process due to the initial input data which can serve as an input for further iterations. This search process can be seen repeatedly in section 5 where first the problem is analysed which leads to alternatives. Due to specific parameters one of the alternatives is chosen and evaluated against the initial performance next.

1.2.2.1.7 Communication of Research The last, yet not less important, guideline is the presentation of the research results to both, technology-oriented as well as management audiences. That indicates that the outcomes or parts of them should, on the one hand, be published in research oriented journals or conferences and, on the other, be conveyed to business leaders able to incorporate the final artifacts or models inside their surroundings. The first part is tried to be achieved by publications in relevant journals, magazines or conferences. The second part of the requirement or guideline is more related to the practically relevant setup of the artifact in realistic scenarios. It is planned to incorporate the final work in, at least, geographic information retrieval systems or other related domain specific applications. Therefore, the communication of the research is necessary in order to help users to set up the index structures in the databases properly as well as to understand the benefits and advantages of the newly developed techniques compared to the currently present ones. As already stated in paragraph 1.2.2.1.4, the research contributions are subject of publications in science oriented research papers. Besides, it is also envisaged to employ the hybrid access structure in real-world scenarios.

This subsection outlines the problem relevance and the connection to the field of design science in information systems research alike. Each of the seven given guidelines for valid and successful research in this application domain can be fulfilled by the above explanations.

1.2.2.2 Regulative Cycle

The second approach of a research method for this work bases on design science in information systems research (see 1.2.2.1). In the case of this thesis, the regulative cycle which is used in design science as nested problem solving [100] can be applied. This theory embeds the basic properties of design science in a larger framework with outer circumstances. Besides the extension of the basic research method, it includes a construct called the regulative cycle.

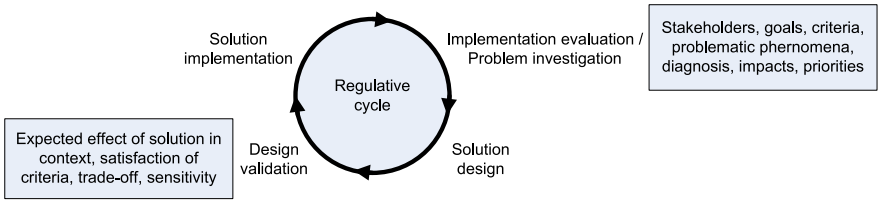


Figure 1.5: Regulative Cycle in Nested Problem Solving [100, page 4]

A conceptual overview over the regulative cycle can be seen in figure 1.5. The regulative cycle is decomposed into four subparts and must be investigated and joined to the problem existing at present. The current problem can easily be connected to the regulative cycle because an iterative approach for the optimisation of hybrid index structures will be chosen.
As a first step, an implementation evaluation or problem investigation must be carried out. The initial analysis of the problem may be executed in diverse investigation variants. Actually, a mixture of impact-driven, problem-driven and solution-driven analysis is applied in this case. The target of the initial implementation of hybrid index structures was to create new access methods which should also be applicable in real world scenarios. After the initial implementation, however, a tremendous reorganisation effort was noticed. Therefore, as the hybrid access structures have influence on actual setups in, e.g., geographic information retrieval or enterprise-wide search solutions, an actual impact in currently inspected real world problems exists. Besides the real world impact, there is also a change in technology which leads to the solution-driven approach. These index structures may not only be set up in relational database systems but may also be ported to other NoSQL or NewSQL approaches like graph databases. Hence, enabling the setup in one particular technology makes the index access methods also suitable for others as most of the approaches store in a disk oriented way. For this reason, the strategies developed for relational databases also apply to, e.g., graph database approaches.

Yet, the examination of other kinds of databases is not subject of investigation in this work but may be applied in the future. The given issues are also problem-driven as index structures were tested in real world applications. This initial setup showed that index structures, in some cases, tend to be inefficient for reorganisation which needs to be investigated. It will be necessary to execute the measurement of not only the problem itself but the detailed information which led to the currently inspected issues repeatedly. As the planned optimisation approach is also iteratively structured, problem investigations will be carried out multiple times and new problems arising from an initial problem statement will arise during the work in progress.
Solution design is the second step in the regulative cycle. The solution for the given problems analysed in the previous steps must be planned now. For this reason, a solution for a current problem needs to be evaluated in order to overcome weaknesses existing at the moment. The solutions may then refer to parts of algorithms inside the reorganisation part of the hybrid index structures. Based on the given measurement values, several influential paths inside a program execution must be selected as candidates for the optimisation. As a consequence, the data must be analysed and one or more solutions for the currently existing problems must be created based on literature studies. Consequently, after the conclusion of this task inside the regulative cycle, the problems determined in the first step to be solved by approaches from literature still exist. However, several different approaches might then be there which need to be evaluated due to the outcome, subsequently.
The next task inside the regulative cycle is the design validation. It includes decisions about, e.g., the validity of a certain design. For this reason, the design alternatives generated for the solution designs must be cross-evaluated in order to figure out one particular final variant. Therefore, additional measurements must be carried out to define the potentially best variant to solve the currently existing and measured problems. A forecast is tried at this point regarding the influence of the particular solution design alternatives. This step generates exactly one implementation variant that can be chosen in order to be implemented in the following iteration of the optimisation procedure for each issue which was shown ensuing from the measurements carried out in advance. It leads, on the one hand, to a set of parts of algorithms or their structures to be changed and, on the other, to a set of solutions evaluated for applicability in the actual access structures to be introduced in the current iteration.
The last step of the regulative cycle is the actual implementation of the chosen variants inside the execution context, which is always the realisation of the selected variant. In this case, the selected adoption is chosen and integrated into the software. The algorithms and data structures inside the hybrid index structure need to be adopted in order to match the given designed solutions. However, the evaluation of the given changes also must be kept in mind. The last phase is the real application of the variant to be realised in this iteration. It concludes one iteration inside the optimisation process of this work.

This conclusive step of the implementation of the selected solution(s) terminates one iteration inside the regulative cycle. Yet, as the main method describes a cycle, the process restarts with a problem investigation. The changes which resulted from a measurement and led to a solution design as well as an evaluation raise potentially new or other problems which must be investigated. Thus, at this point of time, the regulative cycle restarts in order to solve the issues rising in the currently inspected step. One final question inside this process is when to terminate it. However, this must be resolved at the end of the actual work in order to check when the process may be stopped and the optimizations have produced a variant which seems to be "good enough" for the application in real world scenarios. In addition, it may happen that the optimisation probabilities given in the design evaluation of specific parts are simply too low because there are constraints from, e.g., hardware like hard disk, processor or memory constraints which make an optimisation within the software part nearly no longer manageable or only feasible with a huge effort.

1.3 SUMMARY

This chapter has given an introduction to the actual research work. Besides highlighting the content of the research work, the research questions have been outlined based on an initiation to search approaches in hybrid data spaces as well as hybrid index structures in the current implementation variants. In addition to this introduction, a more specific description of the currently used implementation and its problems exists, too. The explanation of the research questions and the general introduction of the approach is also backed by an introduction to the research methods to be used inside the research work. These basic foundations of the work can be found within the work again and will be referred to throughout the entire manuscript.

1.3.1 Phases of the Research Work

The entire work is arranged in the following phases:

1. Initiation Phase
   - Development of a test suite with hybrid access structures on the basis of the H2 database system
   - Definition of the measurement parameters
   - Definition of the input data of the tests:
     (a) Real world data sets
     (b) Standard data sets (Wikipedia, Reuters)
     (c) Synthetically generated data sets
2. Optimisation Phases
   (a) Testing
     - Execution of the tests
     - Analysis of the test data
     - Documentation of the analysis process (which leads to a performance-log)
   (b) Implementation of the selected solution variants
     - Further development
     - Improvement of the algorithms
     - Adoption of the affected structures
3. Finalisation

1.3.2 Structure of the Thesis

This thesis is structured as follows. Chapter 2 describes its theoretical basics, which is followed by the specification of the initial implementation (see chapter 3) comprised by adoptions to a database implementation in Java as well as the realisation of the proposed access methods inside this database framework. As a first contribution, a theoretical model is derived in chapter 4 describing the construction of hybrid access methods and the derivation of one specific parameter of the hybrid index structure used in this work. Its main target is the optimisation of the reorganisation of hybrid index structures supporting multimedia search criteria in chapter 5. First, the setup of the test suite as well as the input datasets and analysis methods are presented. Next, the main work on the algorithm optimisation is executed in multiple iterations which continuously improve the reorganisation performance of the hybrid index structures. The last chapter (6) of this thesis summarises the results where first the algorithms are delineated and an overview of all optimisation iterations is given, which finally shows the tendency of the reorganisation times.

2 THEORETICAL FOUNDATION AND RELATED WORK

This chapter outlines theoretical basics required for reading the subsequent work. Besides the basics, also a summary of the related work and current state of the art approaches is outlined.

2.1 THEORETICAL BASICS

The theoretical basics introduced in this chapter first outline general algorithms which then lead to indexing approaches for relational data as well as multi-valued data sets.
The first references show that technologies in geographical information retrieval require adopted storage mechanisms. They can be implemented in relational databases. Thus, there must exist appropriate access methods to manage data using a combination of relational (or point) values in conjunction with non-relational (e.g. text) data. This introduction is basically given to provide the real world domains and requirements to setup the respective hybrid index structures.

2.1.1 General Algorithms and Information

An approach for building a geo-textual search engine is presented in [38]. The main focus of the search engine is to retrieve security related information from the internet. Therefore, the authors present three different parts used to crawl and analyse the web, store the found data inside an archive and retrieve the data again through a web search interface. Some major challenges of this web search engine approach are also addressed by this paper. The main two challenges addressed in this paper are geo-coding of the documents and the presence of an adopted index structure. Geo-coding means assigning explicit coordinates to a particular document. This implies parsing the full text and analysing it according to its spatial relationships. Adopted algorithms must be supplied in order to parse the document and extract the particular toponymic references from the unstructured / semi-structured texts available in HTML. The search engine approach is limited to news article web sites.
The second major challenge dealt with in the paper refers to appropriate index structures suitable for storing textual and geographical data and searching them simultaneously. Therefore, adopted versions of already present structures are used, namely an adjusted geographic index structure, like the R-Tree, and index structures used in standard information retrieval, like the

inverted index, which can be accessed via a B-Tree. Adopted tree packing methods can be used to improve the performance of the R-Tree. This paper also references the idea of a tree packing method which tries to avoid overlaps in each case to produce an index access method with a well defined worst case retrieval complexity.
For this reason, an appropriate indexing structure which supports storage of data types in hybrid data spaces indexing geographical as well as textual data is necessary.
Furthermore, a more detailed description of the search engine using the respective indexing method is specified in [64]. Besides the utilisation of the hybrid indexing technology, also the process of crawling and the analysis of the search result is described, there. The paper delineating the search engine applies search methods using the hybrid indexing technology. This indicates that it may also be used in a real world scenario.
Another geo-textual search engine approach is called SPIRIT (Spatially-Aware Information Retrieval on the Internet) [56]. It discusses the general aspects and properties of the SPIRIT web search engine architecture. This approach is very similar to the one presented in [38] or [64]. The SPIRIT system consists of a user interface, ontologies, web document collection, indexes, the core search engine, relevance ranking and metadata extraction. The geographical ontology is primarily present to assist the system to disambiguate the user queries entered in textual representation. The web document collection basically represents the crawling and analysing part of standard web search engines (information retrieval) with geographical extensions. Indexes refer to adopted storage management techniques applied to search for textual and geographical parts. However, the authors propose only a very basic approach of full text indexing with a grid based approach for the geographic references. The core search engine part is present for bringing the different functionalities together thus combining the separate components. Relevance ranking applies a joined ranking of textual and spatial predicates. Metadata extraction refers to deriving special data from the given web documents, like geographic names and coordinates as well as appropriate text analysis.

2.1.2 Indexing

Indexing in database systems can be used to enhance the performance of search queries. New technologies have been developed in the last years. Very basic methods suitable for storing atomic numeric and normalised values exist. Besides these normalised values, extensions are given which allow the storage of non-normalised values like texts or geographical coordinates in multiple dimensions. As this work relies on appropriate storage technologies for texts as well as geographical coordinates, some basic definitions are given which use on the one hand classical index structures for accessing normalised values. Further references are added to describe the access to textual storage as well as geographical indexing.

2.1.2.1 Indexing in General

The B-Tree, introduced in [4], is one of the most fundamental access structures in modern relational database management systems. It is used in a wide range of applications for storing and retrieving data in a structured and sorted way. The structure is most suitable for disk resident data. The basic properties of a B-Tree employ a balance, which means that each leaf node appears at the same level relative to the root node. In the original proposal in paper [4], data keys may be stored in arbitrary positions without distinguishing between leaf and inner nodes. Therefore, the B-Tree implies the ordering of the keys in each node. The structure of the nodes always connects a certain key with a particular pointer to a subtree, if there is an inner node, or an original tuple. In the case of a leaf node, the keys point back to the original entries. One basic special case is applied for the first pointer inside a node. Based on the fact that the entries are ordered in some kind of a natural ordering of the keys, the first pointer does neither have key nor value, because it is obvious that each entry in the first subtree must be smaller than the second key - value pair of the currently inspected node. That is why the first element is

omitted and there is only one single pointer referencing a particular subtree present. Besides the general structure, the original paper introduces also algorithms for insertion, deletion and retrieval inside the tree. Today, the B-Tree is one of the most widely used index access methods in modern relational database management systems. In most systems, the B-Tree is also applied to data whenever a primary key condition is set to have a fast and reliable access to the underlying data values, for example in the case of a join.
There are several variants of the B-Tree, most noticeably the B+-Tree and the B*-Tree [61, pp. 481–491]. They employ certain special conditions in order to enhance the B-Tree to some special condition. Basically, the B+-Tree introduces an interconnection of the leaf nodes of the B-Tree in order to allow a faster access to range queries. In order to achieve fast range queries then, only the first (or last) entry must be found which satisfies the lower (or upper) bound condition of the search predicate. Subsequently, the leaf nodes can be followed directly one after the other in order to retrieve each of the entries satisfying the given search condition. Another main feature introduced by the B+-Tree is that entries / values are only posted to the leaf nodes which allows a broader spanning of the internal nodes as more keys may be stored in inner nodes if the values can be omitted. Compared to the B+-Tree, the B*-Tree introduces new concepts of splitting nodes. The minimum fill factor for one node of the B*-Tree is set to $\frac{2}{3}$ of the maximum capacity. Therefore, the B*-Tree, in comparison to the B+-Tree tends to be narrower in terms of amount of siblings in one level indicating a smaller branching factor.

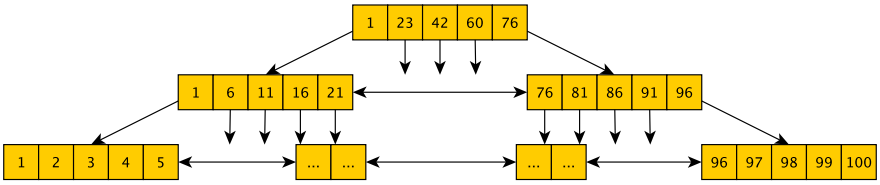


Figure 2.1: Graphical Conceptualisation of a B(+/*)-Tree Node

A graphical conceptualisation including node contents of a B-Tree, actually in the variant of a B+-Tree or B*-Tree, can be seen in figure 2.1. The node and pointer structure as well as the distribution of elements which can be used for sorting on external memory are shown in this figure.
Although the B-Tree can be used to store data on secondary memory (hard disks, ...) very efficiently, it is only suitable for answering single value ($\mathcal{O}(log(n))$) or linear range ($\mathcal{O}(log(n) + m)$) queries efficiently. It does not have the possibilities of working with multidimensional data, efficiently. However, this structure may be used when storing, e.g., numbers, strings or timestamp values.

2.1.2.2 Text Indexing

Text indexing needs to be done in various systems. Document management systems or full text search engines use appropriate storage techniques in order to enhance the access to the underlying texts stored in databases. First, some pre-processing steps must be carried out like term normalisation or stemming [81]. The individual words need to be extracted as a first step in order to put them into the so-called inverted index. Generally, at the beginning, a forward index is constructed which extracts all the individual words for one particular textual document. Afterwards, the inverted index is created from all the items inside this forward index listing the documents and its individual words.
The construction process of an inverted index using three different documents can be seen in figure 2.2. First, the forward index is built using stemming and character normalisation. Thereafter, the inverted index can be created by the mapping of individual words to the respective documents they are contained in.
Inverted file and signature file indexing for text retrieval are explained and compared with each other in [108]. In this paper, not only theoretical assumptions about inverted files and signature

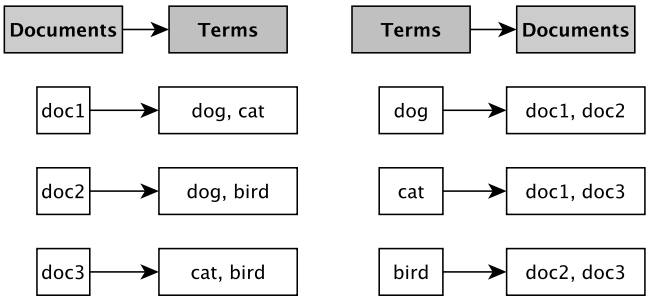


Figure 2.2: Inverted Index Construction

files are given, but also a detailed performance comparison of both indexing techniques is effected. Therefore, the authors apply standard datasets to the structures under test. Inverted files are just inverted index techniques referencing from attributes to objects they are contained in. Signature files, however, assign a fixed size bit string to a certain word using some kind of hashing mechanism. Both of these techniques may be compressed, e.g. by run length encoding for inverted files or probability estimations for bitlists which reduces the size to about 55% of the original size. Comparisons are carried out regarding construction, memory, retrieval and disk space for different kinds of datasets following some kind of standardisation rules which makes them candidates for a proper comparison. The indexing techniques are subsequently tested with a couple of different query types, like ranked queries or boolean queries combining multiple search predicates using conjunction and disjunction. The conclusions state that the standard way of using inverted files is still better compared to the signature files in terms of size, speed and memory consumption.
Besides this information, B-Trees may be used to enhance access (see, e.g. [107]) to posting lists for disk oriented storage. Other options like a PATRICIA tree [77] or hash maps may be applied in main memory. Thus, the B-Tree is used to search the directory of terms and to navigate efficiently to the resulting posting lists which are then intersected with each other to generate the final result set. In order to generate this result set, approaches using pre-sorted lists (for example based on a document id) may be used. In this case, approaches like the no random access algorithm [52, pp. 11:14 – 11:17] can be set up to speed up the generation of lists.

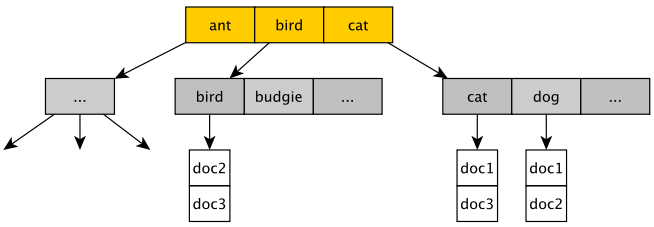


Figure 2.3: Graphical Conceptualisation of an Inverted Index using a B-Tree Directory

A graphical conceptualisation of an inverted index using a B-Tree structure to access the individual words as a directory is given in figure 2.3. When searching for a particular word, the directory can be used to navigate to the respective inverted lists or inverted files and to generate the final result set based on the contents of these files.
Further information about the given texts may also be included inside the inverted index structures. They are mainly used in the list of postings and represent, e.g., the occurrences of entries inside the texts. Using this kind of information, ranking models for text indexing like the TF-IDF model (see e.g. [90, 54]) may be applied after the successful retrieval of the respective instances. This meta-information may, besides the application of ranking, also be used for phrase searches or other kinds of indexing.

Other techniques to store textual data in data stores may also be kept in main memory. As an example for such a structure, the radix or PATRICIA tree [77] can be mentioned. This structure splits textual instances at split points where two respective arrays differ from each other. Therefore, it is possible to guarantee a constant time complexity regarding the length of an object during the retrieval part. The same time complexity can be assured by hash tables in the best case. The worst case time complexity for these data structures is, however, linear. The compressed Trie instead ensures a worst case time complexity of $\mathcal{O}(1)$.

2.1.3 Geographical Indexing

There are multiple approaches for indexing spatial or geographical data. Especially in areas of geographic information systems and the evolving technologies of geographic information retrieval systems, efficient approaches must be created in order to be able to retrieve the desired data, fast. So, appropriate retrieval techniques must be created to get a quick access to these data. Grid files taking a regular partitioning of the underlying space into account are one technique to separate the data objects from each other by assigning them to grid cells and searching them afterwards. This facilitates the refinding of the data objects based on the regular structures they can be found in.

2.1.3.1 R-Tree

The R-Tree [36] is one of the most important spatial index structures currently available. A lot of relational database management systems include this index access method to efficiently explore spatial data stored in the DBMS. The basic structure of the R-Tree is very similar to the one of a B+-Tree or B*-Tree. However, the underlying data and algorithms change in order to apply a spatial data scheme. The R-Tree introduces the concept of the minimum bounding rectangle (MBR), also known as the minimum bounding box (MBB) in some publications. This concept defines the bounds for each subtree pointed to by one element. It stores a lower and an upper value for each dimension to signalise the boundaries of the subtree elements properly. The R-Tree is a balanced structure, similar to the B-Tree, which implies that all leaf nodes appear at the same level relative to the root node. Hence, the length of the path from the root node to each leaf node is identical. Each of the entries is inserted at a leaf node and the structure is subsequently updated to the top or root node. Overflow handling as well as splits and updates of parental entries are also handled in the method of adjusting the tree. Basically, the R-Tree is a spatial extension of the B-Tree with some specialised algorithm approaches.
The basic R-Tree provides multiple ways of choosing a subtree and splitting nodes with different kinds of run time complexities. However, the main focus of these algorithms is on the space utilisation of the different MBRs. Minimising the space utilisation of nodes, however, is not a critical part while creating R-Trees. The main critical part is to avoid overlaps in the nodes. Overlaps result from the dynamic creation of the R-Tree. If overlaps occur, multiple subtrees must be sought when a search query is issued that contains the overlapped region. Therefore, new ways have been investigated to avoid overlaps instead of reducing space. The author provides a full set of algorithms including insertion, searching and deletion. Updating of keys is done via deleting an entry and re-inserting the new key. Two methods of splitting nodes are introduced which, however, only investigate the space utilisation of the MBRs.
The concept of the MBR can be seen in figure 2.4. The minimum bounding rectangle of a higher level element consists of the minimum and maximum extents of the contained children MBRs. In case of point values, the outer point values define the boundaries of the MBR. This concept is similar to the one-dimensional description in B-Trees. In the case of B-Trees, the bounds are implicitly given by the currently inspected value and the following value. Therefore, every value which is, e.g., greater or equal than the currently inspected one and strictly lower than the one of the following element must reside inside the subtree pointed to by the respective element. The MBR represents this information more explicitly. The maximum extents of the area covered

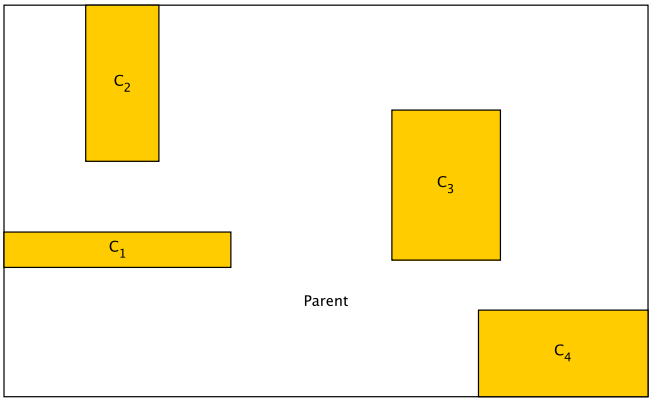


Figure 2.4: Concept of the MBR

by the subtree is given by the minimum bounding rectangle which describes the particular element. The general structure of the R-Tree is nearly the same as the one of the B-Tree. The R-Tree is completely balanced. All values are pointed to by leaf elements. That means that the length of the path from the root node to each particular child is identical in all paths of the R-Tree. Each node, except the root node has between $\frac{M}{2}$ and M elements with M representing the maximum capacity of one node. In realistic scenarios, M is limited by the block size given in database systems. If one node has, at least, $M + 1$ elements, it must be split and results in an additional node which needs to be inserted into the parental node. If this parental node is not present, i.e. the split node is the root node, a new root node is created containing pointers and representations of the respectively new nodes. As already mentioned, the split methods as well as the methods for choosing an appropriate subtree to optimise the tree structure based on certain heuristics are subject of investigation in the further research of the R-Tree. Two of these approaches are described, hereafter.

2.1.3.2 R*-Tree

The algorithms proposed by the R-Tree [36] have been investigated further and according to other conditions. The authors of the R*-Tree propose a new method of splitting by mainly inspecting the overlap conditions of the particular node [5]. The proposed goals are to minimise the overlap and to optimise the storage utilisation. Therefore, new approaches are introduced to select a subtree and to split a node. The algorithm to choose a subtree changes, compared to the one of the R-Tree, in the case of the child pointers of a particular node pointing to a leaf node. If this event occurs, a strategy is chosen which inserts the entry to a subtree whose rectangle needs the least overlap enlargement. Each other case is handled by the standard R-Tree algorithm.
When splitting nodes, one new approach is presented which chooses the axis with the least overlap value. Therefore, the entries are sorted according to the values of the upper or lower bound, respectively, in each dimension. Finally, the axis is chosen having the least overlap value in the particular dimension. There are also possibilities given resolving ties if multiple distributions have the same overlap value according to the area covered by the two entry sets. Besides the new approaches of splitting and choosing an appropriate subtree to distribute one entry to, an algorithm to reinsert data after an overflow is also given which reinserts the entries of a node to the particularly fitting level in order to provide the chance to reorganise the R-Tree during overflow treatments.
The concept of the split method of the R*-Tree can be seen in figure 2.5. Basically, the R*-Tree tries to split the node in half whereas it contains assignments for all combinations which do not favour one node. That means, each of the resulting split nodes comprises approximately half of the entries of the original node. The basic approach takes all configurations into account fulfilling

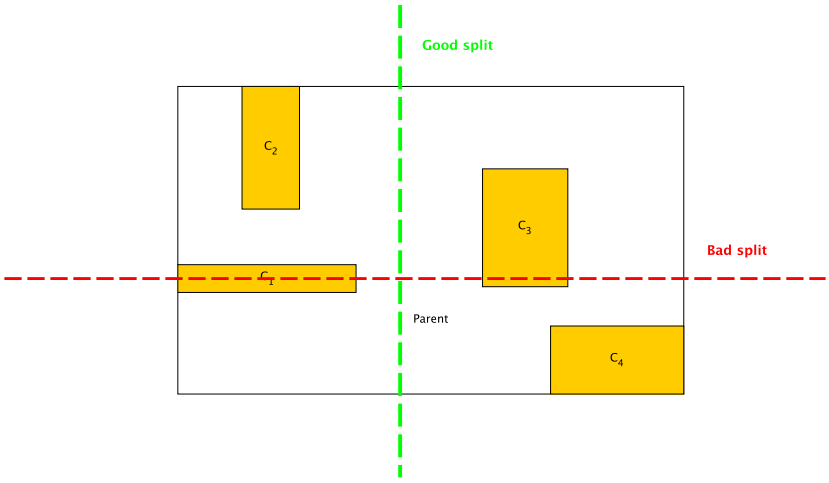


Figure 2.5: Bad Split and Good Split of an R-Tree Node

the criterion that each of the resulting nodes should receive approximately half of the original entries and sorts the elements based on the edges of the geometric objects. After that, the edge and distribution is chosen which has the least overlap. In the best case, the resulting overlap value is zero. However, additional decision parameters are added in case of ties. Figure 2.5 shows two variants of the R*-Tree split candidates. The green "good split" does not cut children MBRs and thus does not produce any overlaps whereas the "bad split" case would cut at least one MBR and thus result in an overlapping area between the resulting nodes. Besides this information, the good split also contains two elements in the two resulting nodes whereas the bad split edge could also produce overlap free children with the restriction that one node would then contain only one node whereas the other would contain three nodes, which leads to an imbalance in the distribution. This is, however, not a valid split as it violates the criterion of splitting the nodes in half. For this reason, the green coloured "good split" case is selected as an outcome, here.
Besides the adapted split version, also a new version of choosing a subtree where to place a new item in is created. This method fundamentally takes the R-Tree base choosing method. However, if there is a node pointing to a leaf node which stores point values, an adopted version is chosen which tries to avoid overlaps on this level.

2.1.3.3 Split Method of Ang and Tan

Ang and Tan present another splitting method for the R-Tree [3]. This split algorithm assumes that the tendency of a particular entry towards the edge of the parental MBR is the most influencing factor for a node to split. Therefore, the authors propose a split method which builds two lists for each dimension. Each entry is assigned to one list in each dimension in a way that it is entered into the list where the distance of one bound to the bounds of the enclosing rectangle is smaller than the other bound in the particular dimension. That is why two lists are built in each dimension which subsequently are used to indicate the split distributions. Thereafter, the distribution is chosen which is most equally levelled regarding the amount of entries contained in the two lists indicating the target distribution of the node. If a tie exists, first overlap values and subsequently coverage is chosen as a split indicator.

The split method based on the approach of Ang and Tan can be conceptually seen in figure 2.6. This figure shows the general approach of the splitting method. The tendency of each MBR to one edge of the parental MBR is taken into account in this case. That means that the minimum

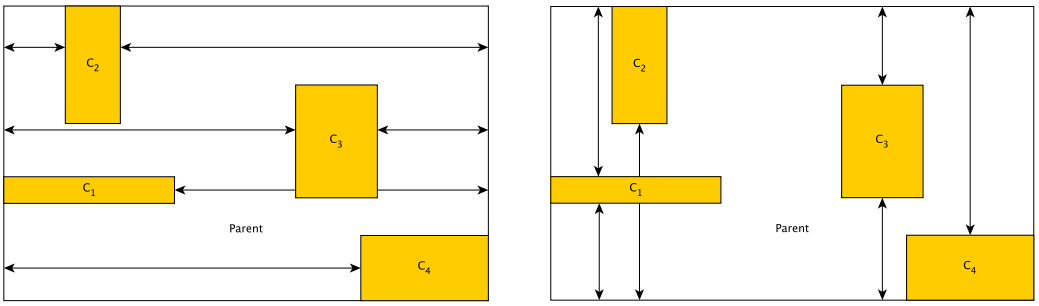


Figure 2.6: Split Method based on Ang and Tan

distance of one object towards the respective edges in each dimension is taken as a criterion for the split. After that, the items are distributed regarding this edge whereas the levelling is also regarded as a criterion. That means, besides the tendency towards the edges, the algorithm tries also to produce resulting nodes having approximately the same number of elements in order to ensure a proper setup of the R-Tree after the split.

2.1.3.4 Tree Packing Methods

Tree packing methods may be applied to optimise the resulting data structure towards some given criteria. They mostly relate to producing overlap free structures, finally. These methods are mainly present to be applied on a fixed set of value objects and are not intended for further changes of the respective structures. Two notable tree packing methods for the use inside the R-Tree are described here.

2.1.3.4.1 Sort Tile Recursive (STR) Tree packing methods are approaches which assume a static data set to work on and pack them into optimised structures. One popular tree packing method for the R-Tree is called Sort Tile Recursive (STR) [69]. This tree packing method has the ability to pack the given set of entries overlap free, at least in the leaf level. Therefore, it separates the entire data set in slices having a regular distribution of entries. So each of the slices has the same amount of entries, which are sorted in the particularly inspected dimension and split into subsets of equal cardinality. This process is repeated recursively in each dimension until the last dimension is reached. There, the remaining entries are distributed to nodes and subsequently returned. This process ensures that the leaf nodes are constructed overlap free. If the leaf level is finished, an adopted algorithm is chosen for the inner levels. This algorithm calculates the center points of the respective multidimensional range values and the original distribution algorithm is applied to these centers. However, static methods assume a static dataset where no new entries may be inserted after construction which means that either, if entries are inserted subsequently, the entire structure degenerates or it is simply not allowed to change entries after construction. This tree packing method has one major drawback. It creates leaf nodes overlap free, however, it does not ensure any overlap free nodes in inner levels of the tree.

2.1.3.4.2 FH Hof Packing Method The FH Hof tree packing method [37] ensures overlap free nodes in each level. Thus, the entire structure created does not have any overlaps at all. Besides this assumption, the tree can also be optimised according to some ratio factor. The time complexity of this solution is a little bit worse compared to STR as the tree packing is a two step

process, here. First, a tree plan is created which generates an optimised structure in terms of the amount of entries inside the nodes. It is ensured that the amount of entries inside the nodes does not vary much. This means that the tree plan is created for optimised space utilisation, as well. After creating the tree plan, the particular entries are distributed to the respective elements recursively inside the tree. Therefore, binary splits are performed on the currently inspected dataset of nodes as long as the optimal distribution is found and each node of the tree plan receives exactly the amount of entries specified before. The goal function for the binary splits is the ratio given in advance which optimises the form of the nodes according to some pre-defined factor. So, basically this tree packing method first performs a bottom up distribution of nodes while creating the tree plan followed by a top down distribution of entries.

2.1.4 Profiling and Software Analysis

This work employs a structured approach for the measurement of the efficiency of software. Therefore, the basics necessary for profiling and static or dynamic program analysis are described now.
One of the most widely used profilers utilising instrumentation is gprof [33]. It can be applied for the static instrumentation of source code and is widely applied in the domain of programs written, e.g., in C or related programming languages. This profiler also comprises an evaluation scheme in the form of flat files using tabular markups as well as a graph representation for the further analysis of the markup data. Other examples of profiling techniques using code instrumentation and dynamic call graph analysis include [95], which also performs an instrumentation based deep profiling of programs utilising C or symbolic computation approaches. Some common problems with these approaches can be solved by means of the analysis framework proposed later in this work. Recursion and multi-threading may also be monitored with the approach easily, depending on the representation of the output files. An approach for profiling including a proper visualisation of the currently executed calls can be seen in or [99]. It tries to visualise the performance of a higher order program by instrumenting the source code and providing an analysis scheme in the program path for the visualisation of live data gathered during the program execution. Hence, the entire process of running algorithms can be monitored with this technique. Besides these live data, it also collects database information to link the analysis results to the source code. Call stack sampling (see i.a. [22, 31]) is frequently used for approximate information about the execution of programs. Compared to a deep profiling run, call stack sampling keeps the instrumentation overhead low and thus does not include much overhead compared to the actual executions of the application in a realistic scenario. Whilst this is a valid approach, it also tends to be imprecise, because by using this technique, the monitoring is executed only occasionally and does not supervise the entire program run. Furthermore, the given articles do not provide any analysis method for the data gathered using call stack sampling.
Analysis techniques for the evaluation of these traces are presented, e.g. in [67]. There, techniques for restructuring the program paths can be found. The introduction of the hot subpaths is similar to an analysis proposed later in this thesis. The difference between the hot subpaths technique and the one employed in this work is that the subpaths solely include one particular subpath, which means that only small parts of an entire execution are inspected. This is not the target of investigation here, because one goal is to find entire algorithms which may be suggested by the paths analysis and not methods or a set of methods somewhere inside the execution. Thus, side-effects may also be monitored for the entire path instead of isolating subpaths.
Further analysis techniques use (dynamic) program slicing to analyse the control and data flow inside a program employing graph based approaches (see i.a. [1]). The slicing approach is more involved in static source code analysis. As a more control-flow and data-centric approach must be chosen to evaluate the target in this case, the dynamic program analysis approach applied in this work better fulfils the requirements of presenting potential candidates for optimisation based on the path pruning analysis in this context.

2.2 RELATED WORK AND STATE OF THE ART

This section outlines the actual state of the art and a general introduction to the hybrid index structure to be optimised in this work. First, a general overview of various present methods is given which is summarised by a descriptive comparison outlining the gaps of existing approaches.

2.2.1 Hybrid Index Structure Implementation Variants

Hybrid index structures, as described in this thesis, are access methods supporting a simultaneous storage possibility inside hybrid data spaces. These hybrid data spaces are set up on a mix of multiple data of different kind. That means that solutions must be found to manage data from relational as well as non-relational types, efficiently. Hence, data structures are introduced supporting the connection to very different data sources, e.g. one or more columns for relational data (including also point values) and one column for non-normalised data types (e.g. texts). The different, currently available, distribution schemes are presented in this subsection. Besides the currently present ones, also the actual structure this work is based on is introduced.

Presently existing hybrid access structures can be categorised by examining their features. There are combined access structures using a primary access structure for one of the inspected dimensions and a secondary one for the other(s). In the context of geo-textual retrieval, they are thus called text-first or spatial-first index structures with respect to the primary and secondary index structure. Examples of these access methods can be found, e.g., in [105, 98, 17, 13, 86]. Although these access methods allow the integration of multiple different data types, they are not regarded as true hybrid access methods but as combined ones. One of the main features of hybrid access methods is the tight integration of multiple access structures supporting the access to hybrid data spaces applying a combined pruning scheme in the course of access method traversal. During the last years, a lot of approaches for hybrid indexing have been proposed. They vary in the combination scheme as well as the basic underlying access methods. Most of them are introduced to support the combined retrieval of geographical and textual data. Examples of "real" hybrid indexing methods are given, e.g., in [20, 59, 70, 101, 30, 43, 18, 39, 40].
Within these hybrid access methods, additional differentiations can be executed based on different features. On the one hand, the type of queries to be handled may be inspected. On the other, also the base structure(s) employed for the construction of access methods may be utilised for the distinction of the hybrid access method types.
With respect to the query type, Chen et al. [16] describe three different query types which are the most commonly used in the domain of hybrid indexing:

1. Boolean *k*NN Query (B*k*Q), which retrieves the *k* nearest neighbours regarding one spatial object in connection with a textual part. In this case, the textual part is only checked for its presence or absence. No ranking on the textual part is applied for these queries.

2. Boolean range Query (BRQ) supporting searches towards a geographical multidimensional range in connection with a textual part, which again is only checked for its presence or absence. This is, generally, the type of queries the hybrid access structure described here is supposed to support, and

3. Top-*k* *k*NN Query (T*k*Q), which is the same as B*k*Q, but also takes into account a textual score implying a ranking function on both, textual and spatial information.

This thesis mainly inspects the BRQ approach supporting a query concept of a multidimensional range in conjunction with boolean textual retrieval. Yet, also access methods with the primary focus of other query types may support BRQ retrieval, too, if the algorithms are changed slightly. Access structures primarily supporting BkQ retrieval include [13, 30, 101]. TkQ is supported by, e.g., [20, 70, 86]. The remaining ones enable BRQ (see [43, 105, 59, 98, 17, 18]), whereas some of the others (see [30, 70, 86, 101]) may be modified to support BRQ queries as well. The target of the access methods strongly affects their design. This indicates that, in some cases, it is not easy to enable support for a different query class if a particular index structure is focused towards one of the given query types.
Another categorisation of the access methods may be carried out on grounds of the basic structures used. In general, "really" hybrid index structures make use of one structure which then tightly integrates another one. Regarding the textual access methods, two different approaches are dominant in the information retrieval field:

1. Inverted Files [107]
2. Signature Files [29]

A comparison of the two approaches is shown in [108]. Inverted index structures invert the assignment of objects and attributes. Thus, the objects do not refer to the attributes stored as fields but the attributes are directly stored pointing to the objects they are contained in. This inversion step is required to split the set-based attributes into parts and index them afterwards (see subsection 2.1.2.2). This procedure enables queries towards elements of the sets. Therefore, it is possible to search for keywords contained in, e.g., textual documents by using the inverted index which presents all objects (documents) containing the keywords. Signature files, however, store the documents as a set of bits. Each term in the vocabulary is represented as one single bit inside these files. For this reason, searches for keywords may be carried out by determining the bit number for this specific case and looking it up in the respective signature files afterwards. In general, these two options are, on the one hand, the dominant methods in textual information retrieval and, on the other, applied to hybrid access methods. Yet, in most cases, only one of them is chosen to manage the textual part of the geo-textual hybrid data space.
As stated above, the textual part is either managed by inverted files or signature files (bitmaps / bitlists). Yet, for the spatial part, there are additional categorizations:

1. R-Tree (or variants) [36]
2. Grid files [79]
3. Space filling curves (SFC) [50, 78]

One of the dominant disk-resident access methods in databases is the R-Tree (see subsection 2.1.3.1), which is a multidimensional extension of the B-Tree. R-Trees, including their variants, are one of the most widely used techniques for the management of geographical and spatial data and work efficiently, at least for low dimensional data spaces. Yet, in standard geographic information (retrieval) systems two-dimensional data with longitude and latitude are generally expected. These data spaces may still be efficiently handled by R-Trees. Other possibilities of spatial data management include grid files. They normally split the underlying data space in equally-sized grid cells so, that each of the cells occupies the same dimensions. This data management technique is very good suited for equally distributed data but may struggle with other distributions, like normal distributions, because some cells are more populated than others in this case. Yet, grid based storage is still used quite frequently in database systems, especially, if the data may be assumed to be arranged appropriately. One crucial setting for the setup of grid based storage is the grid size. In grid based access methods, the quantity of elements in one grid cell varies, particularly, regarding different settings of the grid dimensions. Thus, they

may serve to smooth internal differences in the data distributions. The third technique employed as underlying spatial management variant is the application of space filling curves (SFC). One example is the Hilbert curve [50] describing a curve to fill multidimensional data spaces. It is used, e.g., in the Hilbert R-Tree [57], which distributes the data objects according to the Hilbert curve. Another variant of a SFC is the z-curve [78] utilising a different approach to fill the space. In general, these curves are applied to map multidimensional data space to one dimension. Afterwards, data pre-processed using the SFC approach may, e.g., be stored inside a conventional B-Tree [83].
Some of the hybrid indexing variants are described in more detail in the following subsections.

2.2.1.1 Bitlist Hybrid Index

One basic method for hybrid geo-textual indexing is presented in [39]. It describes the use of a mixed inverted index and R-Tree approach for frequent terms. In case of infrequent terms, the approach simply applies a search on the inverted index with a subsequent filtering step which removes the entries not fitting to the spatial part of a particular query. Appropriate algorithms are presented used to insert the document postings to the particularly adequate position inside the entire hybrid tree structure. Outer circumstances and parameters are also discussed. There are, basically, two parameters which must be set in advance that control, when a certain term is removed from the inverted index and inserted to the hybrid part of the R-Tree. The other parameter limits the length of the bitlist. However, this artificially set parameter is only valid when considering main memory implementations. Generally, in a standard database environment there is a certain block size which limits the amount of entries that can be stored inside a particular node. Therefore, the block size limitation also influences the length of the bitlist and thus, this parameter cannot be set arbitrarily in such an environment. The approach assumes an R-Tree which has been statically constructed by a tree packing method, presented in [37]. If the number of words overflowing the upper bound parameter for terms referencing documents becomes too large to be stored in a single bitlist, the geographical index structure is duplicated. Thereafter, the second geographical structure stores the entries which overflow the upper bound parameter.
This structure has been developed inside a project which forms the basis of the current work. For this reason exactly this structure is taken into account for optimisation inside this work. It is referred to as **B**it**l**ist **H**ybrid **I**ndex (BlHI) in the following.

2.2.1.2 Id List Hybrid Index

Another hybrid index structure is proposed in [40]. It makes similar assumptions as the one submitted in [39] and described in subsection 2.2.1.1. However, there are differences in terms of the handling of entries. Instead of the bitlist, an id list as well as an adopted inverted index variant are attached to each particular element. The id list stores frequent terms that reside somewhere in the subtree of the currently inspected node. In comparison to the bitlist oriented structure, which only utilises the frequency metrics for distinction in an initial step and placing all items to secondary inverted index structures attached to leaf elements of the hybrid R-Tree, this structure is set up to make the decision at each level. Hence, secondary inverted index structures can be found at each level of the hybrid R-Tree and the frequency metrics are decided locally. For example, if the limit is set to 200, all items appearing in more than 200 documents are placed inside the first level whereas all items emerging between 200 and 400 times are put to the second level of the R-Tree etc. Hence, smaller and more specialised secondary inverted indices occur in this structure.
This is continued until the leaf level is reached. There, no further distinctions can be made and all entries which have not been placed are put to the secondary inverted index structure there. In case a term entry is moved to a deeper level its id is entered into the id list and thus marking

that it resides in some deeper level in the tree. If the id list is full, a "least recently used" strategy is applied for further filling.
The search algorithm would then first search in an initial inverted index for either document entries or ids to be found inside the hybrid part of the index. This is basically the same strategy as used by the bitlist hybrid index. Thereafter, the ids found in the initial inverted index are looked for in the hybrid R-Tree together with the spatial region to be determined. Therefore, the id list is searched and, if no particular entry is found there, the search continues inside the B-Tree attached to a particular element. During searches, the id list attached to each hybrid R-Tree element may also be filled, which results in a caching functionality for this structure. Finally, all outcomes fulfilling the spatial as well as the textual part of the query are collected. Thus, the id list and bitlist structure are very closely related to each other and mainly differ in the way of treating the term to document links stored in secondary inverted index structures. Whereas all of these structures can solely be found at the leaf level of the bitlist structure, they may be present in multiple levels for the id list structure. It is referred to as **I**d **L**ist **H**ybrid **I**ndex (ILHI) in the following.

2.2.1.3 Further Hybrid Index Approaches

The structure described in [104] is also bitlist based. Its focus is to find a pre-defined number of *m* tuples of keywords in a *d*-dimensional feature space with the diameter as small as possible. For this reason, the algorithms for querying and inserting are strongly biased towards this use. However, the general structure is still an R-Tree with textual extensions using bitlist references.

The content of [30] depicts the construction of a hybrid index structure called the IR^2-Tree (Information Retrieval R-Tree). It stores each of the entries at the leaf level and references the particular words attached to R-Tree nodes using a bitlist structure signalising the presence or absence of a particular word at a node element. A second version of the IR^2-Tree is also presented which applies super imposed coding of subtrees if a particular term does not appear inside an entire subtree thus overcoming the restriction of the limited size of the bitlist available for storing the indicator of presence of a particular word. However, this structure is optimised to answer top-k queries in a GIR system or for general applicability besides top-k domains.

Paper [72] gives a short overview about currently available indexing mechanisms in spatio-textual indexing. The index proposed in this paper relies on a combination of a space filling curve and an inverted index.

Comparisons of combined index structures comprised by inverted indices and R*-Trees are done in [105]. There, different combinations of R-Trees and inverted indices are given. Three combinations are presented and evaluated. The R*-Tree and Inverted Index double index is simply too slow, because in case of queries, it can be generally taken for granted that it generates two separate result sets which must be intersected at the end. Thus, there is a large overhead of merging probably producing only a small amount of results, finally. The second option is First Inverted File then R*-Tree. This option is clearly the best, because specialised spatial structures are checked after having found them in a textual context. But as there are many duplicates, this structure simply uses too much space on the disk. The last discussed option is First R*-Tree then Inverted File. However, this structure probably searches large parts of the spatial structure while totally ignoring the textual part which leads to the fact that many subtrees are followed contributing only a small amount or no results to the final result set.

Paper [43] describes a GIR system which is capable of storing and retrieving entries in a geographical and textual way. The authors first describe the comparison of already present index structures which include a combination and chaining of multiple index structures, mainly the R-Tree or one of its variants and an inverted index. Yet, this combination either does not have enough performance or simply requires too much space. Therefore, the authors provide a new index structure, called the KR*-Tree, which combines the inverted index with an R*-Tree in order

to be able to search for textual and spatial components simultaneously. However, this KR*-Tree still stores the keywords in a list separate from the R-Tree only referencing back to the nodes the particular keywords appear in. That is why an inverted list is built which references from the particular keyword to the nodes where the keyword is stored in. But the lookup for one of the terms might still take a long time as each of the keywords must be looked up on each node during a query which might result in a performance issue.

The contents of paper [98] describe the different indexing approaches used in the SPIRIT search engine. The authors propose a two level indexing scheme, either pure text, spatial primary, text primary or text indexing with spatial post-processing. Most of the structures are defined as grid based without making use of the dynamic adoption of, e.g., an R-Tree. So, the approaches are only evaluated among themselves without taking into account any other indexing approaches.

2.2.1.4 Empirical Validation of Hybrid Index Structures

There is an empirical validation of hybrid index structure types (see [16]) which performs experiments using real data sets on different kinds of hybrid index structures. These access methods are separated regarding the target queries they are capable to process. The categorizations of queries are the same ones as described before (BkQ, BRQ andTkQ).

The evaluation of these particular query schemes as well as index structure schemes which are also separated in multiple categories based on the chosen implementation variant in this case is executed using three pre-defined data sets with very different properties. Besides the respective query scheme, input data and data structure classification, typical database related parameters, e.g. buffer size, page size or also the number of objects inserted are evaluated. Due to the evaluation of BkQ a grid based method is clearly favoured. However, grid based methods are simply too strict for a lot of applications and may not be generalised for most applications. During the evaluation of BRQ and TkQ, an approach which also uses the application of the separation between high and low frequently used terms and an R-Tree as a base structure outperforms the other methods. This study principally shows that R-Tree based dynamic spatial indexing structures deploying a text separation that utilises additionally augmenting structures and, potentially, the storage in inverted or signature files may surpass other kinds of approaches in this area. Therefore, the chosen structure presented in [39] (see subsection 2.2.1.1) should be a sufficiently good strategy for the retrieval task in the domain of spatial keyword queries. That is why the main decision regarding the implementation of the basic index structure towards the bitlist based hybrid access structure in combination with R-Trees seems to be reasonable. However, owing to previous performance measurements, the structure works well as to search performance but still struggles in the reorganisation part inside realistic application scenarios.

2.2.2 Comparison

A listing of tested features and environments is given in table 2.1. It shows a list of the most commonly applied hybrid indexing techniques separated by the base storage structure categories. Each of them is mainly described regarding the tuple construction or general structural features. In addition, most of the access methods are only tested on disk and not implemented in a realistic database scenario. The most important difference between the database and disk oriented management is that a realistic database environment adds more dependencies and other management structures for recovery and logging. The main focus of the tests is to evaluate the retrieval performance. Therefore, it is obvious that the search procedures are well described in most of the papers and are also the principal target of empirical evaluations. The insertion, update or reorganisation procedures are either left out of the description in total or only outlined in a very high level way and not delineated in detail. Additionally, nearly none of the structures performs measurements regarding the inserts. Two

Structure Variant	Model	Environment			Algorithm		Query Type
		RAM	*Disk*	*DB*	*Search*	*Insert*	
R-Tree based							
SKI [13]	✗	✗	✓	✗	✓	✗	BkQ
(CD)IR-Tree [20]	✗	✗	✓	✗	✓	✓*	TkQ
(M)IR^2-Tree [30]	✗	✗	✓	✗	✓	✓*	BkQ (BRQ)
KR*-Tree [43]	✗	✗	✓	✗	✓	✗*	BRQ (BkQ)
IR-Tree [70]	✗	✗	✓	✗	✓	✓$^+$	TkQ (BkQ, BRQ)
S2I [86]	✗	✗	✓	✗	✓	✓$^+$	TkQ (BkQ, BRQ)
WIBR-Tree [101]	✗	✗	✓	✗	✓	✗	BkQ (BRQ)
IF-R*-Tree / R*-Tree-IF [105]	✗	✗	✓	✗	✗	✗	BRQ
BIHI [39]	✗	✓	✗	✗	✓	✓*	BRQ
ILHI [40]	✗	✗	✗	✓	✓	✓*	BRQ
Grid based							
SKIF [59]	✗	✗	✓	✗	✗	✗	BRQ
ST / TS [98]	✗	✗	✓	✗	✗	✗	BRQ
SFC based							
SF2I [17]	✗	✗	✓	✗	✗*	✗*	BRQ
SFC-QUAD [18]	✗	✓	✓	✗	✗	✗	BRQ

Table 2.1: Comparison of Hybrid Access Methods

*: no tests are executed / only description

$^+$: only measurements / partially

approaches ([86, 70]) perform a measurement of the insertions as well but not very detailed. Therefore, these measurements may only serve as an estimation.
This table shows the gaps of currently present access methods. Most of the hybrid index structures are only evaluated on disk in simulation based environments or even in main memory. Only one of the approaches [40] is implemented in a realistic database (PostgreSQL) and evaluated regarding the performance measures, there. Yet, this structure also only performs evaluations regarding the retrieval performance and does not take any insertions into account. Table 2.1 shows that there are a lot of different access methods for hybrid data spaces. Most of them are well evaluated as to retrieval performance. In addition, many of them are also implemented in a disk based environment. Yet, there is an obvious lack of integrating them into a realistic database kernel. For this work, mainly methods supporting the query type BRQ are interesting, because the target is to support typical queries in a geographic information retrieval system or in enterprise content management systems.
Thus, in total, it can be seen that the currently existing gaps of the state of the art access methods for hybrid data spaces are:

1. Implementation and evaluation of access methods in realistic database scenarios. Most of the data structures are realised and tested in main memory or on disk. While the disk oriented testing approach is already suitable for giving hints about the performance in databases with the indication of block access and disk reads, it is not sufficient to outline this part of the performance because databases use other kinds of abstractions to the file system. Furthermore, there are other kinds of data accesses than direct block accesses. Especially, the task of the management of recovery and write-ahead-logging must be done manually in databases and includes additional block accesses.

2. Detailed and elaborate description of the insertion procedure in order to keep the effort low when reorganising the hybrid index structure in case of insertions, updates and deletions. Retrieval processes are described explicitly in the present approaches. Yet, the construction and modification of the access methods is delineated rather superficially and not (intensively) evaluated.

3. Necessity of a generic model description to derive functional and worst-case efficient storage structures. In the current approaches, no reference is given of a generic model used to derive worst-case time and space constraints for present access structures. Although multiple different datasets are applied to the evaluations of the access methods, no direct conclusions are given about theoretical limits of the hybrid index structures. The construction of the index structures is only described from a performance point of view and not from a theoretical one giving insights about worst-case complexities and size constraints.

2.2.3 Summary

Although there are many approaches which model hybrid access structures, they are mostly evaluated in synthetic simulation based environments. The main target of the validation is focused towards the analysis of the retrieval efficiency. Even comparisons regarding the efficiency are outlined, e.g. in [16]. However, most of the introductions of structures only list the basic data structure model and omit the introduction of appropriate algorithms for modification purposes. High level analyses and descriptions of the algorithms exist, which primarily aim at providing insights of the general construction procedures. Yet, there is a lack of detailed descriptions of the reorganisation algorithms. The proposed algorithms also work well in simulation-based environments which do not require the proper treatment of database specific attributes like the generation of write-ahead log files and efficient buffer management. This indicates that high-performance retrieval structures are present which, unfortunately, are mostly only validated regarding their query behaviour and do not focus on the proper construction and manipulation. In addition, there are many different approaches of constructing the hybrid access methods in terms of the combination of different features. Thus, a generalised model and the understanding of connections between the data structures is not given either, yet. Currently, also modelling approaches for index structures in general are given, e.g. in [45], which focus on the definition of workloads and not on the derivation of worst case complexities to deduce final models for access structures. This thesis is focused on the derivation of a generalised model and the introduction of efficient reorganisation algorithms of hybrid index structures supporting multimedia search criteria to close the gaps discussed above.

3 INITIAL STATE OF THE IMPLEMENTATION

This chapter describes the basic implementation used for the hybrid index structure. Mainly, already present approaches and their instantiation are characterised here. This initial implementation is performed to port the existing methods to a working environment in a realistic database scenario. From the already present methods of implementing hybrid access methods, a structure using an R-Tree augmented with bitlists is used (see [39]). As it is quite similar to the one depicted in [40], nearly everything from the construction to the optimisation approaches also applies to it. They have been implemented and validated in this work. The original implementation had been done in the PostgreSQL[1] database in a previous project. However, based on a paradigm change, the decision was made to switch to another database system written in Java. Several different Java based database systems have been tested (including Apache Derby[2], HSQLDB[3] and H2[4]). The final decision was taken to use the H2 database system to port the index structures to. Some of the main reasons are a good code structure and a well defined index access method definition interface.
Yet, some adoptions had to be made in order to support the hybrid indexing scheme at the database. In addition to the general implementation to be optimised, these adoptions are specified in this chapter.

3.1 DATABASE EXTENSIONS

The H2 database system as such only supports a few index structures. They include a B-Tree or a hash table oriented approach for indexing data in main memory. However, as hybrid index structures are planned to be implemented inside this database, several extensions must be done in order to enable this particular indexing approach.
Initially, all database index structures are loaded from the library containing the database. To permit a more flexible approach, the index structures should be loadable from external places. Another issue, here, is the comparison approach. As basically only items are compared inside the currently existing database which support a natural ordering, merely comparison checks indicating a sorting can be executed on the stored values. Regarding multidimensional access

[1] `http://www.postgresql.org/`, accessed 2013-08-02
[2] `http://db.apache.org/derby/`, accessed 2013-08-02
[3] `http://hsqldb.org/`, accessed 2013-08-02
[4] `http://www.h2database.com/html/main.html`, accessed 2013-08-02

structures, this comparison approach is not sufficient for, e.g., multidimensional point values. Therefore, a mechanism for comparing other types of data which, probably, cannot be ordered in some kind of natural ordering in one dimension must be employed, as well. Furthermore, specifications for these two mechanisms must be provided in order to signalise to the database the presence of the need of these particular techniques. That means that e.g. certain language grammar parts must be introduced in order to support these features.
With reference to the comparison mechanism, an external approach is introduced which, based on a predefined interface, is able to carry out value related comparisons and data type modifications. This allows the index structures to stay independent from concrete implementations of data type whilst enforcing some predefined structures composed in the interface.

3.1.1 External Loading of Index Structures

The external loading of index structures is realised by using the Java reflection API[5]. Some prerequisites are introduced for procuring the possibility to load index structures from external places. First, a basic interface is defined which each of the customised index structures is supposed to fulfill in order to get loaded from an external place. Second, some meta information must be placed inside a specific library. The index structure implementations are supposed to be placed inside a well defined folder on disk. The last precondition is the definition of a constructor which must be implemented by all index structures in order to be loaded.
Two additional tables are introduced inside the basic database storing information about the respective index structures. The access methods as well as operator classes must be registered in these tables before they may be used.

Table	Column	Type	Meaning
INDICES			
	ID	int	Primary Key
	NAME	varchar	Symbolic name of the index structure
	FILE	varchar	File name to find the index in
OPCLASSES			
	ID	int	Primary Key
	NAME	varchar	Symbolic name of the operator class
	FILE	varchar	File name to find the operator class in
	INDEX	int	Index reference (Foreign Key)

Table 3.1: Table Definition for Index Structure Dynamic Loading

The basic definition of the newly introduced database tables required for the external loading process can be seen in table 3.1. The indices and opclasses table are very similar except that the opclasses table has an extra foreign key reference to the index table. Each operator class is connected to one particular index structure. More than one operator class may be associated to an index structure. Therefore, there is a $1 : m$ relationship between indices and opclasses. Symbolic names for both structures are also introduced which can be used inside the customised SQL grammar introduced in order to be able to load the index structures to a specific table.

```
1 CREATE { [ UNIQUE ] [ HASH ] [ indexType ] INDEX
2 [ [ IF NOT EXISTS ] newIndexName ] | PRIMARY KEY [ HASH ] }
3 ON tableName ( indexColumn [,...] ) [ USING opclassType ]
4 [ WITH additionalArguments ]
```

Listing 3.1: Modified CREATE INDEX Command for Enabling Custom Index Structure Loading

The syntax of the customised SQL command for creating index structures on database tables

[5] http://docs.oracle.com/javase/tutorial/reflect/, accessed 2013-08-15

can be seen in listing 3.1. This command has two basic modifications regarding the original one. On the one hand, it created an `indexType` to be set up and, on the other, it enables a `USING opclassType` statement at the end. These two modifications are established to address the dynamic loading procedures. Yet, some additional prerequisites for using this modified command exist.

3.1.1.1 Basic and Interface Definitions

The customised index structures (contained inside libraries as *.jar files) must be placed inside a folder called "index" which must reside relatively to the startup path of the application. The structure of the *.jar file as well as the contents are described later on. Additionally, the information must be registered inside the database tables in order to get the command to run properly.

```
1  package de.fhhof.iisys.index.h2;
2
3  public interface CustomIndex
4  extends org.h2.index.Index {
5      /**
6       * assigns the given operator class to the present index
7       * @param opClass the operator class to be set to the custom index
8       */
9      public void setOpClass (de.fhhof.iisys.index.h2.OperatorClass
10      opClass);
11     /**
12      * creates a dummy {@link de.fhhof.iisys.index.h2.Page} valid
13      * for the current index which can be used to read in the contents
14      * @return the dummy {@link de.fhhof.iisys.index.h2.Page}
15      */
16     public de.fhhof.iisys.index.h2.Page createDummyPage ();
17     /**
18      * indicates to the index that the rebuild process is started
19      */
20     public void setStartRebuild (org.h2.engine.Session session);
21     /**
22      * indicates to the index that the rebuild is finished
23      */
24     public void setEndRebuild (org.h2.engine.Session session);
25 }
```

Listing 3.2: Interface Definition for Custom Index Structures

Listing 3.2 shows the basic interface definition for custom index structures to be implemented by external index access methods. The custom index interface has four essential methods. Besides these methods, there are also some functions to be overridden by implementing classes from the `Index` interface provided by the base H2 database implementation. The `setOpClass` method sets the operator class associated with the currently used data type to the index. Each index structure may supply customised database page implementations. Therefore, the process of using a dummy page which then may read index specific pages is used to read load data from the respective pages. This functionality is introduced in order to be able to modify or create customised database pages. The other two functions marking the begin or end of the rebuild procedure are mainly important for applying packing to some index

structures. The R-Tree, e.g., has the ability to be pre-packed using, for example the sort tile recursive (STR) [69] or the FH Hof Packing [37] approach. Each of the customised index structures to be loaded from external places must implement this basic interface.
Besides implementing the interface, a proper base class can be chosen in order to fulfill the goals of the respective index structure. Very often, when implementing disk oriented index structures, the page index can be chosen, here. The base definition of a specific class, in this case, the R-Tree, can be found in listing 3.3.

```
1  public class RTreeIndex
2  extends org.h2.index.PageIndex
3  implements CustomIndex {
4      // ...
5      /**
6       * default constructor of the R-Tree called from the H2 Database
7       * @param table the {@link Table} back reference
8       * @param id the id of the index needed to initialize the
9       * base index
10      * @param indexName the name of the index
11      * @param columns the columns to be indexed
12      * @param indexType the {@link IndexType} reference containing
13      * many informations about the index
14      * @param create indicator whether or not to create the specific
15      * index (or to just load it from the disk)
16      * @param session the {@link Session} to be used
17      * @param opClass the operator class to be used for
18      * specific operations
19      */
20     public RTreeIndex(Table table, int id, String indexName,
21         IndexColumn[] columns, IndexType indexType, boolean create,
22         Session session, OperatorClass opClass) {
23         // ...
24     }
25     // ...
26 }
```

Listing 3.3: Basic Class Definition of the R-Tree Index

Besides the basic class definition, the necessary constructor called by the reflection based instantiation approach is also listed there. The respective parameters can be seen in the JavaDoc like description. The information given in the respective fields is supposed to be persisted on the disk to be reloaded on demand when the index is required. This is, in particular, the case when the database is reopened. These basic implementation specific prerequisites must be given in order to make one particular index access method loadable from the disk.
It is recommended to use Apache Maven[6] for compiling and packaging the index structures. One example pom file for controlling the maven build as well as making specific additional entries can be seen in listing A.2. The probably most important entry in this case is the definition of the startup class using the maven-jar-plugin in the manifest entries. Inside the MANIFEST.MF file of the resulting jar file this configuration creates an entry specifying the startup class. It is the one implementing the `CustomIndex` interface.
The procedure for loading a custom index structure which is, basically, the same for operator classes as well, can be seen in figure 3.1. Based on the changes in the create statement the parser first identifies, that there is a request to a customised index structure. It triggers the

[6] `http://maven.apache.org/`, accessed 2013-08-15

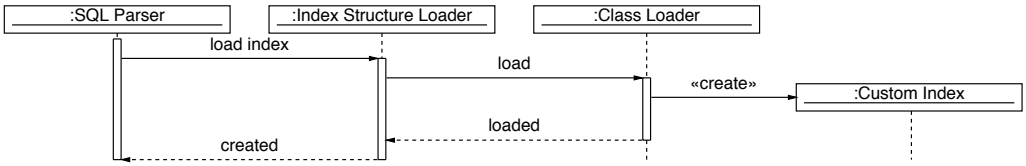


Figure 3.1: Sequence Diagram of the Dynamic Loading Procedure

index structure loader. The symbolic name for an index structure as well as an associated operator class name are identified from the SQL command. Due to this information, the given tables (see table 3.1) are queried for the resolution of symbolic name to file name. This loader is aware of the structure of the *.jar file as well as the basic interface definitions. Therefore, it loads the *.jar file and inspects the manifest file for the startup class which is then loaded via reflection and returned to the SQL command. Thereafter, the table may add the index and instantiate it (again using reflection techniques) by issuing a call to the respective constructor. The constructor definition to be fulfilled by each customised index structure can be seen, as already mentioned, in listing 3.3, lines 20 – 22. This is the basic process to instantiate and assign a certain index structure to a table where actions to the particular tables and tuples can be managed. Using this approach, there is an easy way to store index structure implementations at external places and load them on demand.

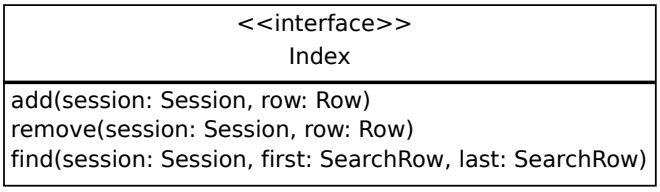


Figure 3.2: Basic Interface Definition of an Index Structure

The basic interface definition of an index structure used by the H2 database is shown in figure 3.2. Only the most important functions for modifying or querying the index structure are listed here. The row as well as the search row class are abstractions to tuples containing values. Owing to the columns handled these values are indexed there. However, the values are still abstract definitions containing only basic functionalities. The goal of an index structure, in general, is to stay abstract from concrete implementations of the underlying data types. Therefore, it is important to work only on the basic value definition and not to use predefined data types. The basic value class hierarchy can be seen in figure 3.3. In this picture, only an

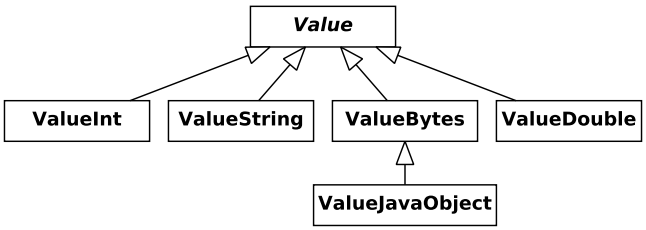


Figure 3.3: Basic Value Class Hierarchy

excerpt of the class hierarchy is displayed. Basically, the index access method implementation is supposed, in order to stay data type independent, to work only with the base value class. Java objects may also be stored directly there and be accessed via the byte array representation using a serialisation approach. To apply customised comparisons and modifications for the data

types, an adopted way to modify or test the data types must be created. This is done by using the operator class interface definition.
The three methods described in figure 3.2 outline the base functionality for an index structure to fulfill. The `add` method allows adding new rows into the index. It is either called when inserting new values into the database or when modifying rows using the update command. Updating rows is generally realised by deleting the old row, or at least marking it as deleted and inserting a new version, later on. Removing a row is implemented by passing it to the index which then needs to find the reference to the row and remove it or mark it as deleted. The find operation is executed by the use of a cursor concept. This cursor is then returned as an immediate result having a state where a call to the `next` function generates the first result. Therefore, it is important to place the cursor in a state where it resides "before" the first row to be found. Besides these operations, also additional functionality is required by the index structures. However, most of the remaining methods are mainly available for administrative tasks like checking the size of the index or querying the amount of tuples stored there for consistency reasons.

3.1.1.2 Page Management

One important feature for a disk resident index structure is the ability to read or write pages from or to the disk, respectively. The H2 database already comprises functions which may be used for these tasks. However, in order to allow customised database pages to be loaded or stored on disk, additional mechanisms had to be created which allow customised storage techniques. During the reading process of particular pages, the page store, which is the main administrative component for interacting with the disk in this version of the database system, determines some basic information from the pages written in advance. Therefore, according to the storage structure, one particular page must follow some conventions in order to be properly stored or loaded to or from the hard drive.

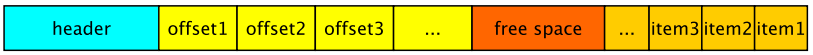


Figure 3.4: Basic Setup of a Page Inside a Database System

The basic setup of a page which may be stored inside a database system is defined in a similar way as it is demonstrated in figure 3.4. This storage model is inspired by the N-ary Storage Model (NSM, see, e.g. [82, pp. 327 – 333]) where especially the storage of variable length types is inspected, here. The header information may store fixed size and predefined data blocks which are, generally, representative for all pages of the same type. This header information stores global information about the pages and is representative for one page type. In many cases, e.g. the B-Tree, one index structure contains exactly one page type (including some variants) which shares exactly the same header information. Therefore, the page store may, on grounds of the header information, load the specific page into main memory. This is not true for hybrid index structures because these structures are composed by multiple, potentially very different, page types. Yet, some information stored there always resembles the same structure and is thus characteristic for the index structure stored.
Normally, the approach to fill the remaining free space of the database pages is done via specifying offsets the real data items are retained in. The offsets are stored at the beginning of the page filling it from the beginning to the end. The real data item payload, however, is recorded at the end of the page. This procedure fills the data pages from both sides. The offsets are filled from the beginning to the end whereas the data item payload is stored from end to front. This indicates that the free space of the database pages is available between the offsets and the items. If the space for the offset field and the items would overlap, this indicates that the page is full and some additional actions must be taken. The procedure of putting the offsets to the front and the items to the end of the page helps to avoid many copy operations in some kind of array representing the bytes within the page if items are added sequentially. The items

just need to be added one after the other while the offset array grows from the one side of the page and the item array from the other. This indicates that the aftermentioned mechanism of filling pages is practical. Therefore, it is used by many database management systems supporting disk resident page stores, like PostgreSQL or the H2 database.

Figure 3.5: Header Information in the H2 Database

The header block storing the meta information of one particular page inside the H2 database can be seen in figure 3.5. The basic definitions given by the database system are coloured cyan.

Structure	Type	Meaning
type	byte	page type identification
parent	int	parental page number (if any)
checksum	short	bytewise checksum for consistency checking
index id	int	identification key for the index structure
custom	variable	customised information available for the index structures

Table 3.2: Page Header Information Definitions

The storage structures given in figure 3.5 are detailed in table 3.2. The page type is an identification for the respective page to be loaded. On the grounds of this, the storage manager can decide about the procedure how to load the respective page. The custom index definition supplies a value of 9 here to enable the custom index structure loading mechanism. This is a convention introduced during the adoptions for the support of external index structure loading. The parent field may be used to include hierarchical information inside one particular page. Not all page types use this field, it is however imposed by the H2 database system itself and thus cannot be changed. Yet, it may also be used for saving other types of data. The checksum field is used to store a checksum calculated from the data of the respective page in order to ensure the data consistency. Index ID stores the identification number of the associated index structure of the H2 database. They are basically the only structures which must be stored inside the header information as the page store uses them to determine the proper loading mechanism. The remaining header fields may be filled arbitrarily by the index structure in order to save respective meta information. A B+-Tree could, e.g., store a reference to the right or left sibling inside the header information in order to represent a linked list based storage here.
For being able to handle customised loading methods of pages, there is the method `createDummyPage()` inside the `CustomIndex` interface definition given in listing 3.2. For most applications, it is sufficient to use a singleton pattern to create the dummy page as this page type is solely responsible for deferring the loading mechanism. This dummy page is used to read custom index pages thereafter. Thus, the page store will call this method and let the dummy page do the further work if a page type of 9 is encountered. A derived page class all other page types are supposed to inherit from is also created which additionally adds the method `read()` to the basic definition of a page (see listing 3.4).

```
1  package de.fhhof.iisys.index.h2;
2
3  public abstract class Page
4  extends org.h2.store.Page {
5
6          protected Page () {
7          }
8
9          /**
10          * reads a data page using the dummy definition
11          * @param idx the {@link Index} to read from
12          * @param data the {@link Data} to be used
13          * @param pageId the pageId
14          * @return the {@link org.h2.store.Page} read
15          */
16         public abstract org.h2.store.Page read
17             (org.h2.index.Index idx, org.h2.store.Data data, int pageId);
18         /**
19          * reads the contents of a page
20          */
21         public abstract void read ();
22 }
```

Listing 3.4: Abstract Class Definition of a Page

Basically, the two additional methods shown in listing 3.4 enhance the functionality of the particular page to be able to read something from the disk because this is not intended by the original implementation. Yet, each of the implementing subclasses of the `Page` class had to read the data from the disk. The first function returning a `Page` object is primarily used inside the dummy page which is supposed to read and instantiate the proper page representation. Actually, each of the pages that can be used in customised index structures to be implemented should also derive from the `Page` class as given in listing 3.4.

3.1.2 Operator Class Definition

As already discussed, the index access method itself should be as independent from underlying data types as possible. Generally, the access structure is not supposed to know anything about the data it stores at all. This makes sense as a lot of data types should exist which can easily be integrated to be stored inside an index. This independence helps, for instance a B-Tree, to be able to index very heterogeneous data types, whilst actually storing only one at a time. The only precondition for the mechanism of staying independent is the presence of certain interface functions. A B-Tree, for example, does only require the functionality to compare particular values in order to sort items. The basic value class definition already supports this kind of comparisons. However, more complex data structures also require more complex comparison or data modification operations. As an example, the R-Tree needs functions which compare the equality of two points or operations which include the modification or comparison of rectangular regions. In this case, one option is to work on these data types directly. However, this bears the risk that the structures thereafter are too specific and cannot work at all on other kinds of data with similar attributes. In order to avoid this situation of becoming too specific, each index structure is allowed to declare some kind of operator class definition. This is a basic interface to be implemented by concrete realisations which care about the data types stored. Therefore, an R-Tree, for example, may require functions for instantiating a range type from a point definition or the enlargement of this particular range by a point or another range. Using the basic value definitions in conjunction with an interface allows the concrete modification of data types

without having to know the exact type which is worked on. Hence, each customised index structure comes in conjunction with (at least) one operator class realisation which implements the basic interface of an operator class exposed by the index access method.

```
1  package de.fhhof.iisys.index.h2;
2
3  public interface OperatorClass {
4          /**
5           * retrieves the serialized version of the original
6           * representation
7           * @param orig the original representation
8           * @return the serialized form derived from the
9           *  original representation
10          */
11         public org.h2.value.Value getSerializedForm
12             (org.h2.value.Value orig);
13         /**
14          * retrieves the original version of the serialized
15          * representation
16          * @param v the serialized version
17          * @return the original form
18          */
19         public org.h2.value.Value getOriginalForm
20             (org.h2.value.Value v);
```

Listing 3.5: Operator Class Interface

Listing 3.5 displays the basic operator class interface each of the operator class interfaces valid for particular extending operators for respective index access methods. Operator class definitions of concrete index structures are supposed to extend this interface and add additional functionality used by the concrete index. The two methods of generating an original or serialized version of a value are needed in order to (de-)serialise the values to create more compact representations for storage because the original Java serialisation produces, in some cases, very large binary objects which could impose problems while storing them into pages. This is due to the potentially large amount of metadata added during the serialisation process in order to be able to properly deserialise the data later.
The concrete operator class definitions are also stored in the subfolder "index" inside *.jar files. The loading mechanism is comparable to the one used for index structures. The opclasses table (see table 3.1) contains a foreign key reference to an index structure. This conjunction states the index the particular operator class can be applied to. The structure of the operator class including metadata inside the jar file is very similar to the one used for the index structures.
One additional convention had to be done to the operator class implementation which cannot be manifested by the use of an interface definition. This basically relies on the instantiation process. As the operator class, similar to the index startup class, must be instantiated via a reflection mechanism, one particular constructor must be present by convention. The empty default constructor is required in this case to enable proper instantiation of concrete operator class definitions. If this constructor is not given for the defined startup class, an instantiation is not possible and thus the external index structure loading mechanism is supposed to fail at this stage.
An overview of the connections between the database, index structures and operator classes is shown in figure 3.6. This figure also represents some of the concrete implementations of index structures added in this work.

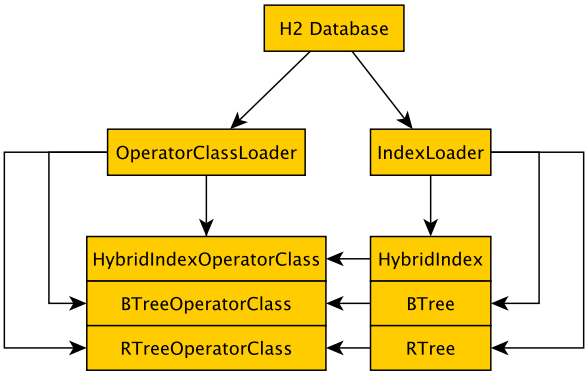


Figure 3.6: Implementation Details of Loose Index Structure Coupling

3.2 INITIAL HYBRID INDEX IMPLEMENTATION

The initial hybrid index structure implementation, which is generally the foundation of this thesis, is described in this section. It is probably worthwhile to describe the initial implementation in some detail as the following optimisation iterations change a lot of internal details of the index. Therefore, it makes sense to focus on some details inside the entire construction as well as the particular algorithms employed in the reorganisation and search process of the index structure. The hybrid index structure is composed by multiple more simple access structures. These structures are used in a predefined sequence in order to distribute the items to the places they fit to based on the distribution scheme given in the hybrid index. Therefore, first an overview is given about the entire dependencies of the particular storage structures and then the particular individual structures are described in a more detailed way.
The index structure basic implementation bases on a previous work which was implemented inside the PostgreSQL database system. It is first ported to the H2 database in order to execute the optimizations inside this system. The implementations used here are generally a one to one port of the original C sources to a Java based version inside the H2 database. As the two systems are very similar, at least regarding the basic definitions, it is easily possible to port the original implementation to the new system. It must be noted that the prototype in the PostgreSQL database was implemented to ensure the enhanced retrieval capacity of the hybrid index structures. It was implemented without any considerations with respect to the insertion or reorganisation performance. Hence, it uses very basic implementations of the actual storage structures.

3.2.1 Hybrid Index Structure Conceptual Overview

A conceptual overview of the hybrid index structure is given in figure 3.7. This is the first base version of the index which already has all storage structures (except one) included. It is, generally, described in [39].
The hybrid index is supposed to provide fast access to data used in geographic information retrieval systems. As these systems include textual as well as spatial data, the index structure itself is supposed to support fast access to data composed by textual and spatial parts. The queries for this structure are targeted towards a set of query keywords as well as a spatial range. Therefore, the combination of an inverted index with an R-Tree is chosen for this task. In order to be able to search in the two domains simultaneously, the R-Tree is augmented with a bitlist representing the presence or absence of a certain term from the inverted index. Besides this basic definition, a differentiation is made between high and low frequently occurring terms. This helps limiting the length of the bitlist which is stored alongside with the R-Tree content

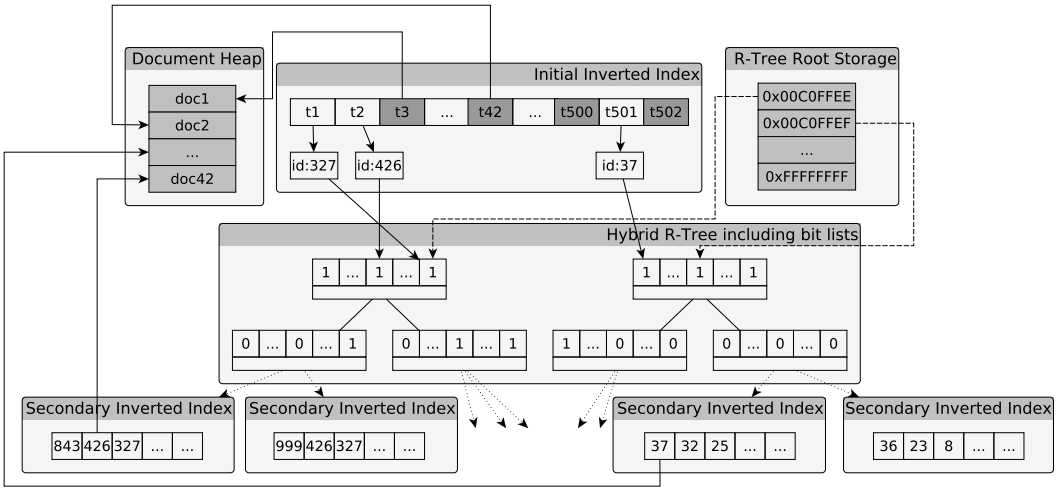


Figure 3.7: Component Overview of the Hybrid Index Structure

elements. An arbitrarily user defined limit called "*HLimit*" is introduced, here. It is strongly application dependent and may be adjusted based on the application requirements. It decides whether a certain term is regarded to be high or low frequently. Currently, there is no option to determine this limit automatically for arbitrary types of inputs. It primarily serves for separating the input terms in two sets and only regarding the really frequent terms for being further processed inside the index.
When a new item is inserted, the first storage structure affected is the so called "document heap" (see the left top side of figure 3.7). This storage structure serves for storing the point values associated to one particular document. A document in this case is a record to be stored in the database table the index is set up on top. The document heap saves only the points contained in one particular document as they are, potentially, not directly processed further and required as such later on, hypothetically. An arbitrary amount of points associated to one document record may exist. Therefore, it is important to keep in mind that already the points associated with one document may need to be split in order to fit in one page. The reference of the first entry stored inside the document heap is thereafter used for further references to the document itself. That means that there is an indirect access to the documents, further on, as the original reference as back link to the user table is only saved inside the document heap. The document heap is implemented as a doubly linked list. In order to fill the particular pages as "good" as possible, there is the possibility to create partitions of the points to avoid sparse pages in this case. The result of placing the point values to the document heap is a positional information which represents the reference to this particular instance. The reference always points to the first instance if, e.g., the point array had to be split to avoid empty space in the pages. This reference is worked on further in the subsequent structures and does, after placing the item to the document heap, represents the positional information as well as the main reference to the newly inserted item. It is important to keep this in mind as all references from inside the structure always refer to this instance later on and do not directly refer to the original database tuple. That means that the outcome in this case is a pair of a page identifier where the item was placed and a relative offset inside this particular page representing the placed item. These references are further on used as keys representing pointers to the original tuples throughout the entire access structure.

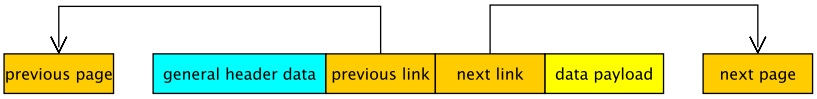


Figure 3.8: Document Heap Page Setup

The basic setup of a document heap page is shown in figure 3.8. The general header data are already described in section 3.1.1.2. This page type implements a doubly linked list where one page always references the previous and next pages in a chain. So, the possibility to follow the links from front to back or vice versa is provided. The data payload consists of the real tuples.

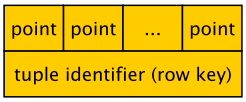


Figure 3.9: Document Heap Tuple Overview

The construction of the particular tuples can be seen in figure 3.9. It is basically constructed from a point array and a tuple identifier. The tuple identifier is the back link to the original database record in the table the index is set up on. This is the only place inside the index structure where this tuple identifier is used. At other places, the page id and position id inside the respective document heap page is used.
The next step in the insertion process is to work on the initial inverted index. This index structure is set up to, on the one hand, work on the textual components of the stored tuples and, on the other, to decide whether to process a particular tuple further. It uses the arbitrarily user defined limit "*HLimit*" . This limit is used to decide whether one particular word entry inside this index structure occurs low or high frequently. It is a corpus dependent setting which needs to be adopted based on the textual components of the entire corpus added to the database or the index, respectively.
Usually, the insertion procedure is done as in a general purpose inverted index. However, some adoptions are made in order to be able to support the hybrid index based features.

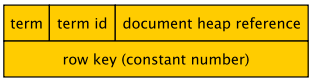


Figure 3.10: Tuple Construction for the Initial Inverted Index

The tuple setup for the initial inverted index can be seen in figure 3.10. It is saved as a Java object which is serialised to the database pages, manually by writing data into a byte stream, in the initial implementation. This holds for all described tuple types in the initial implementation. It is set up by the basic term to be saved, a term id and the document heap reference which identifies one particular element from the document heap. The row key which must be set for all `SearchRow` instances is fixed to a constant number in the initial implementation and therefore not used at all. The term refers to the actual word inserted here. The term id is assigned if one particular term is referenced more than "*HLimit*" times. Then all items equalling the term are removed, too and only one instance is kept together with the term id. This also ensures that the initial inverted index only needs to save the low frequently occurring terms. The high frequently occurring terms are still indexed there, but only once for the purpose of retrieving the term id which is used for subsequent operations.
Therefore, the conceptual overview of the insertion operation at this stage is to form a `SearchRow` which has the setup as described in figure 3.10. Each term occurring in the given text is handled separately here. The constructed tuple is then used for searching for the presence of the particular term. If the term is already present, each occurrence is counted and afterwards evaluated, provided that the reference count exceeds "*HLimit*". If a term id is already set, here, the counting step can be skipped as only one reference is stored there, if the overflow happened, before. In case the overflow occurs when the currently inspected item is inserted, a (index) globally id as a numeric integer value is assigned to the term and all but one occurrences in the initial inverted index are removed.
The terms which either overflow the limit or have already overflown it before are put into a list which is subsequently handled. A non-empty list is an indicator for the index structure that a further action must take place inside the hybrid index structure. Therefore, the initial inverted

index exists, on the one hand, for the handling of low frequently occurring terms and, on the other, for deciding whether to process some terms further. It must be noted that the term ids have two important meanings. On the one hand, it is clear that if a particular term id is set inside the initial inverted index, this exact term must be processed further, later on. On the other, the term id is assigned to avoid having to treat terms any more in further steps of the index structures. This helps because the term ids are of fixed size (integer values of four bytes) whereas terms, generally, may have an arbitrary number of individual characters. In the case of unicode characters, each character might have more than one byte allocated resulting in the situation that the four bytes of the given integer are already overrun with two characters inside one word. Therefore, as a side effect, the amount of storage required by the index structure can also be lowered by using codes instead of the full terms. Another, probably more important feature in this case is that the term ids can be used to represent specific bit numbers which are used in the further step of the insertion procedure.
The next step in the insertion procedure is the treatment of the overflowing terms. As already discussed, they are kept inside a list of elements where each term id points to a list of positional information referring to the document heap items. This overflow list is supposed to be treated by the actual hybrid index component, which is an R-Tree augmented by a bitlist. With each term corresponding to a term id, this term id is used as bit index inside the bitlist indicating the presence or absence of a certain term inside a given subtree of the R-Tree. That means that, in this stage, combinations of textual and spatial information is built in order to be able to provide simultaneous searches on heterogeneous data types. It must be noted that the hybrid structure may not only be applied to textual structures in combination with spatial ones. The underlying structures may be exchanged providing broader application domains. A B-Tree could be augmented to supply, e.g., searches on data joined by time ranges in combination with sets of entries.

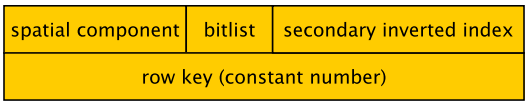


Figure 3.11: Tuple Construction for the Hybrid R-Tree

The tuple definition for the hybrid R-Tree can be depicted from figure 3.11. Once again, the initial version stored a serialised version of a Java object containing information about the spatial component (e.g. a point), the bitlist and a reference to a structure called "secondary inverted index". This structure is described later on.
The spatial component for the R-Tree is a point which is loaded from the document heap based on the references stored in the overflow list. The bitlist gives information about the presence or absence of particular terms identified via the term id inside the subtree pointed to by the currently inspected R-Tree element. As a first step in the initial implementation, the R-Tree is filled with all points inside the given overflow list whose references point to the document heap. Therefore, the document heap references must be loaded and all points from the documents stored in advance must be inserted to the augmented R-Tree. That means that tuples of the given structure must be created and put into the R-Tree. First, the information inside the bitlist or secondary inverted index fields stays empty, because the distribution of terms and documents to the respective subtrees is performed in a further step.
The next step in the insertion procedure is the distribution of the entries to the respective subtrees they fit in. For this purpose, the R-Tree is traversed recursively and the entries are put to the places where they fit spatially. The list of entries built during the insertion of the terms to the initial inverted index is supplied to a method which is responsible for generating the list of elements valid for the respective element. Therefore, each of the entries pointing from a certain term id to the entries from the document heap is checked to find out whether at least one point reference inside the document heap entry fulfils the spatial criterion. In the case of an inner component, a check is carried out to see if at least one point is contained inside the range which describes the coverage area of the R-Tree element. If so, a sublist is generated which points from the term id to the document heap entry. In the case of a point entry, the check is

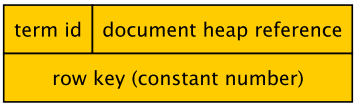


Figure 3.12: Tuple Setup of the Secondary Inverted Index

performed for the equality of the two inspected entries. The respectively generated sublist is then passed recursively to the distribution algorithm which it works until a leaf node is hit.
The leaf node entries point to the final destination of the entries. This storage structure is called the secondary inverted index.
The tuple setup of the secondary inverted index is shown in figure 3.12. It can be seen that the basic tuple construction is very similar to the construction of the initial inverted index (see figure 3.10). The essential setup consists of the term id to be stored as well as a reference to the document heap entry. The row key stays constant in this case. The implementation of the secondary inverted index is exactly the same as the initial inverted index. The only thing that changes is the internal state using a different kind of treatment of the concrete data objects stored inside the tuples of the respective storage structures.
As a last step, the recursive distribution algorithm reorganises the bitlists in order to update the bitlists of the respective elements with the aim to execute the necessary updates to be able to refind the placed tuples later on. That means that each term id is set (if not previously set) to the bitlist of the respective R-Tree element after placing the elements to the subtree.
At this stage, the initial inverted index contains all terms from the respective tuples. Some of them, which means the frequent ones, have a specific term id set which indicates that they refer to the hybrid index. Each of the documents containing at least one frequent term is also distributed in the hybrid R-Tree. These documents are distributed spatially through the R-Tree and are placed to the secondary inverted index which is the final destination of the entries and contains entries which point from a certain term id back to the document heap reference. Therefore, the documents can be found by following the initial inverted index to the hybrid R-Tree to the secondary inverted index.
However, one important property still must be mentioned. Each of the described storage structures is implemented as a serialised version of a Java object in the initial implementation. This serialised version is deserialized for each operation or comparison executed on the object to be handled.

3.2.2 Index Structure Setup and Operator Class Definition

The hybrid index structure supports a basic definition of documents following a certain setup. The document to be indexed must follow a specific definition. This definition is specified inside the operator class interface defined by the hybrid index.

```java
package de.fhhof.iisys.index.hybrid.base;

/**
 * basic interface for the operator class to be
 * used in the hybrid index
 * @author ckropf2
 */
public interface HybridIndexOpclass
extends OperatorClass {
        /**
         * the R-Tree opclass associated with the current
         * hybrid index
         * @return the R-Tree opclass associated with the
         * current hybrid index
         */
        public RTreeOpclass getRTreeOpclass();
        /**
         * retrieves the words as {@link List} from the
         * original document
         * @param doc the document which is to be composed
         * into the distinct word parts
         * @return the {@link List} of words stored inside
         * this document
         */
        public List<Value> getWords (Value doc);
        /**
         * retrieves the points from a given document
         * @param doc the document to get the points from
         * @return the {@link List} of {@link Value}s representing
         * the points
         */
        public List<Value> getPoints (Value doc);
        /**
         * retrieves the max docs flag (how many docs in
         * maximum may be referenced from within one level)
         * @return the maximum amount of docs which may be
         * referenced from one of the entries in the initial B-Tree
         */
        public int getMaxDocs();
        /**
         * retrieves the amount of elements allowed to be stored
         * inside the R-Tree
         * @return the amount of elements to be stored in the R-Tree
         */
        public int getRTreeElementCount();
        /**
         * retrieves the target length of the bitlist
         * @return the target length of the bitlist
         */
        public int getBitlistLength();
}
```

Listing 3.6: Hybrid Index Operator Class Interface

An excerpt of the hybrid index structure operator class interface can be seen in listing 3.6. Only the most important methods are catalogued, there. Some additional management functions are

also included which are, however, omitted in this listing. The operator class allows to get a reference to the R-Tree operator class which is used to compare or modify the elements from the R-Tree. Mainly a range and a point implementation are used in this case. There are possibilities to receive a list of words and a list of points which are associated to the document changed in this case. The words are modified in the initial inverted index where an additional internal operator class is applied. Besides these functions, also management functions are listed here. The retrieval of the max docs integer value represents the artificial limit *HLimit*. This threshold indicates whether a certain term is regarded as being frequent or infrequent. The other two getter functions influence each other. Only one of the two parameters may be set here. That means, the R-Tree element count and the bitlist length are mutually exclusive. Either the number of elements inside the R-Tree or the bitlist length may be set because they influence each other. The amount of elements stored inside one R-Tree node affects the length of the bitlist as each R-Tree element contains a bitlist. As a fixed quantity of memory is available for one node, enlarging the number of elements inside the node reduces the length of the bitlist. An implementation for the given operator class is available by applying a document implementation composed of a text and points. In order to instantiate it in this way, a table is required which has a document column.

Table	Column	Type	Meaning
DOCUMENTS			
	ID	integer	Primary Key
	WORDS	txt	normalised terms
	POINTS	array	ARRAY of points (point objects)
	DOC	document	computed column from words and entities

Table 3.3: Definition of the Documents Table

An example for a table structure which can be used for instantiating the hybrid index structure can be seen in table 3.3. The column "doc" is a computed column from the words and the entities. This is the column where the hybrid index is supposed to set up on.

```
1 CREATE bitlisthybridindex INDEX ON documents (doc)
2 USING bitlisthybridindex_document_opclass
```

Listing 3.7: Hybrid Index Instantiation SQL Statement

Besides the overview about the given structures as well as the algorithms which are subject of change inside this work are described in the following sections. The initial implementation is changed over time during the respective optimisation iterations.

3.2.3 Individual Storage Structures

After describing the entire reorganisation process and the hybrid index structure as such, it is worthwhile to investigate the individual structures involved in the hybrid index. The main index structures which affect the hybrid index structure are the R-Tree which is augmented by a bitlist as well as the inverted index. The inverted index is accessed via a B-Tree and is used multiple times. Therefore, an implementation is chosen that allows easy modification of the concrete algorithm without having to change the B-Tree as such.
Each of the individual storage structures is implemented in a distinct project. This enables individual testing of the index structures, on the one hand, regarding stability and, on the other, by addressing certain issues and test features or potential improvements individually without having to set them up inside the hybrid index framework.

3.2.3.1 Inverted Index / B-Tree

The inverted index used in this implementation is accessed via a B-Tree. That means that the B-Tree forms the directory to navigate to the inverted index entries. They are primarily comprised by terms. Each of these terms points to an inverted list containing the documents with the respective words.

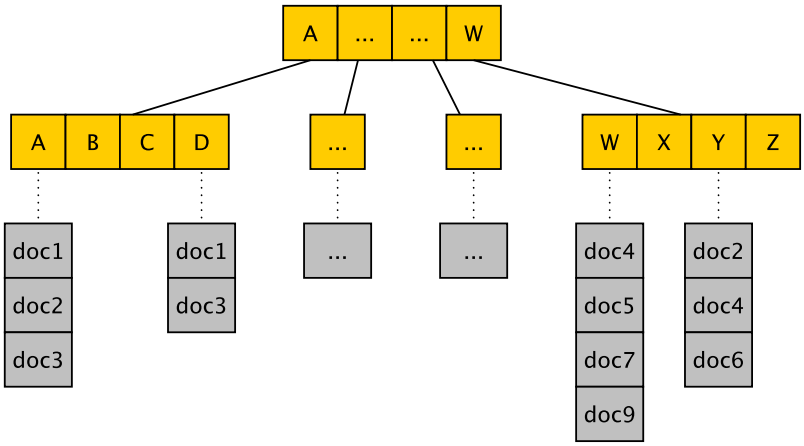


Figure 3.13: Inverted Index Accessed via a B-Tree

A conceptual overview about such a structure is displayed in figure 3.13. The node coloured yellow represent the B-Tree elements serving as a directory to navigate to inverted index elements. Besides the navigation support, the B-Tree also allows sorting in secondary memory. The elements coloured gray inside the given figure represent the inverted lists pointed to by the respective word entries. In some cases (as given in figure 3.13), the list of document references is sorted. It may be useful to augment additional information inside the documents. These information portions may include the occurrence of a word inside the text (e.g. for phrase searches) or the number of occurrences inside one given text. This extra information may, subsequently, be used for, e.g., ranking or retrieval in a vector space model approach.
The initial implementation of the B-Tree uses a B+-Tree approach which stores pointers between the nodes and is therefore suitable for range searches.

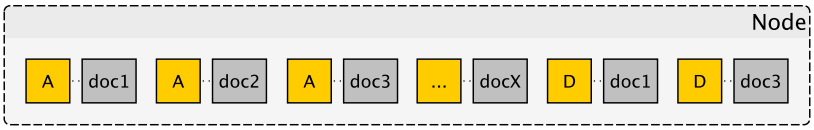


Figure 3.14: Initial Node Structure

The initial B-Tree node structure is shown in figure 3.14. There, the storage of the document references is of special interest. They are stored directly as pairs inside one given node. That means that the inverted lists are put directly inside the B-Tree nodes. Each document reference is stored as a pair of entries, there. The term to document mapping is done directly inside the node and many occurrences of one particular term may take place, there.
The B-Tree uses a standard implementation, as described in [4]. Extensions regarding storage structures are made to implement a B+-Tree. Compared to the original implementation, this structure stores the actual data portions at the leaf nodes. The B-Tree is suggested to store the data in inner nodes, too. The main benefit in this case is that the directory stays smaller as only keys are indexed inside the inner nodes. As range searches may just descend to the first key inside a leaf node which fulfils the search criterion and subsequently follow the chaining until the upper bound of the range does not include a certain key the chaining of the nodes is another benefit. This structure can be used as a stand-alone structure in order to test it outside the hybrid index.

3.2.3.1.1 Operator Class The B-Tree is used from an inverted index structure. This inverted index is also implemented stand-alone. It is required to leave this structure independent of concrete data structures, too, because it is employed in different application domains. The initial inverted index applies the terms from the textual documents to index them. The secondary inverted index instead stores the term identifiers as keys. Therefore, two different implementation variants of the operator class must be used to support the storage in both types.

```
1  package de.fhhof.iisys.index.h2.btree;
2
3  public interface BTreeOpclass extends OperatorClass {
4          /**
5           * compares two LITERAL values decomposed in advance
6           * @param v1 the first {@link Value} to be compared
7           * @param v2 the second {@link Value} to be compared
8           * @return the comparison value in integer representation
9           */
10         public int compare (Value v1, Value v2);
11         /**
12          * converts the given {@link Value} to a {@link Value}
13          * which can be used in inner nodes
14          * @param leafVal the leaf {@link Value} to be
15          * converted into an inner {@link Value}
16          * @return the inner {@link Value} converted
17          * from a leaf {@link Value}
18          */
19         public Value convertToInnerValue(Value leafVal);
20 }
```

Listing 3.8: B-Tree Operator Class Definition

For supporting proper sorting, the basic B-Tree operator class solely uses two functions (see listing 3.8). The first one compares two literal values. It returns a comparison result with the following possible values:

- < 0 if $v1 < v2$
- > 0 if $v1 > v2$
- $= 0$ if $v1 == v2$

The second function is available to convert a particular leaf entry into an inner value. In some cases, the structure of leaf entries may be different from inner values. Therefore, this function may either implement something to change the entire structure or just return it again if the same value may be applied in inner nodes as well as in leaf nodes.

3.2.3.1.2 B-Tree State The operator class allows customisations regarding specific data types. For this reason, providing a specific implementation for a certain data type, e.g. numeric values or strings, enables the B-Tree to store exactly this data type. However, using the B-Tree structures in different application specific domains sometimes requires also different types of additional functions.
The additional functions include the following:

1. `setRoot` for setting a new root node
2. `getRoot` complementary function for `setRoot`
3. `getAdditionalInformation` providing access to additional information
4. `getAdditionalFunctions` giving access to additional functions

Several others are also included which, however, are not important enough to be described. Setting and retrieving the root node must be adapted as the root node or information about the root node must be stored in different places according to the use of the structure. For example, applied by the initial inverted index, the root node is stored in a structure called `MasterPage` which saves additional management information about the entire index structure. When the B-Tree is used inside the hybrid R-Tree, the place to store the information about the root node is a leaf element of the R-Tree as an internal place of information. The entire implementation of the B-Tree should, however, not change according to the usage. Therefore, the state interface must be implemented by a concrete class which implements all the relevant functions that might change according to the usage. Thus, getting or setting the root node is thus clearly dependent on its application. The other two functions `getAdditionalInformation` (see 3) and `getAdditionalFunctions` (see 4) deliver further storage or modification information for the B-Tree index.
The additional information which can be retrieved by using the respective function, is information stored globally in all nodes. This allows, e.g., a back link from the root node (or each node, respectively) to the storage structure the B-Tree is referred from. This might be an important piece of information if a B-Tree node is moved or the root node changes and must be written to the original storage structure. Because of this information, the `setRoot` function may place the information about the new root node directly into the R-Tree leaf element which points to the respective B-Tree.
The additional functions object which may be retrieved via `getAdditionalFunctions` can be utilised to get access to further management functions. They include some operations which are mainly necessary, if the B-Tree is used as a basic structure for a hybrid index. The main use of the functions is described in subsection 3.2.3.2 because this structure provides exactly the same functions and is the main focus in this thesis.

3.2.3.2 R-Tree

Similar to the B-Tree, the R-Tree is also implemented in a library of his own. Therefore, it can be tested standalone and finally be integrated to the hybrid index. As some properties of the R-Tree must be determined externally, it is also worthwhile to implement it in an external library, which can be used to be integrated to the hybrid index afterwards.
Compared to the B-Tree that stores single values (like numbers or strings) inside one node, the R-Tree saves a combined key representing a range in multiple dimensions which contains all children elements. The leaf elements in this case describe point values without any spatial extent. The concept of the minimum bounding rectangle was proposed by the original paper [36]. This is a structure which contains the least extended bounds of all the edges of the children elements. Figure 3.15 shows a parent node with four children, again represented via the MBR. The general tree structure is very similar to the one of the B-Tree. As the R-Tree is generally a multidimensional extension of the B-Tree, this seems obvious.
The R-Tree implemented here is basically structured according the implementation which can be seen inside the original paper. However, some additional functionalities are given. The functions to choose a proper subtree for placing a particular item as well as to split a particular node are the subject of investigation in some subsequent publications. Therefore, a mechanism is introduced to switch between the respective implementations.
Besides the original versions of choosing a subtree and splitting a node which, generally, optimise the spatial area covered by the respective node elements, two additional

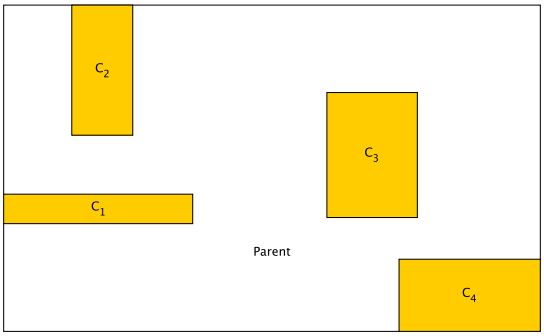


Figure 3.15: Concept of an MBR for an Internal Element

Parameter	Condition	Values	Meaning
choose	no packing / ratio	(rtree | rstar)	Choose Subtree Algorithm Selection
split	no packing / ratio	(rtree | rstar | angtan | rplus)	Split Algorithm Selection
packing	no choose / split	(str | fhhof)	Tree Packing Algorithm
ratio	no choose / split & packing=fhhof	d<number> d<number> ...	Ratio of Tree Nodes for FH-Hof Packing Algorithm

Table 3.4: Overview Over R-Tree Optional Parameters

implementations are given. The R*-Tree variant [5] tries to optimise the node overlap by properly sorting the elements. It also supplies a new variant of choosing a subtree to find out where to place the new item in. Another implemented version is the split algorithm introduced by Ang and Tan [3]. For evaluation purposes, these three methods are supplied together with the R-Tree base implementation and may be combined and tested in order to investigate the respectively "best working" variant. The R+-Tree splitting variant [92] is also implemented which, basically, additionally allows downward splits to reorganise the R-Tree also in lower directions to split the final version of the tree free of overlaps.
Furthermore, the R-Tree has also been extended to allow pre-packing to optimise the R-Tree regarding a certain criterion. Therefore, a constant distribution of the elements is assumed which does not change any more. These approaches are often referred to as "Tree Packing". Two variants can be applied, here:

1. The sort tile recursive [69] approach splitting the elements in equally levelled portions and building the elements upwards, later on and

2. a version totally free of overlaps (also in inner nodes which cannot be ensured by the STR approach) version [37].

The parameters may be set by the use of the arguments inside the SQL statement to create the index. In this case, the `WITH` part of the create index statement (see listing 3.1) can be used to supply the arguments.
The potential additional arguments which may be supplied to the R-Tree can be seen in table 3.4.

```
CREATE rtree INDEX idx_rtree ON points (pt)
USING rtree_point_opclass
WITH (choose=rstar, split=rstar);

CREATE rtree INDEX idx_rtree ON points (pt)
USING rtree_point_opclass
WITH (packing=fhhof, ratio=d1d1);
```

Listing 3.9: R-Tree Create Statement with Different Combinations

Two different configurations for creating an R-Tree are shown in listing 3.9. The first one sets up an R-Tree index on column pt of table points with R*-Tree choose subtree and split algorithm. The second one supplies the FH Hof packing approach on a two-dimensional point column using a ratio of 1 : 1 of the node edges.

3.2.3.2.1 R-Tree Operator Class Definition The basic operator class definition of the R-Tree includes more functions compared to the B-Tree operator class (3.2.3.1.1).

```
package de.fhhof.iisys.index.h2.rtree.operator;

public interface RTreeOpclass
extends OperatorClass{
        /**
         * generates a range {@link Value} from a point {@link Value}
         * @param point the point {@link Value}
         * @return the range {@link Value}
         */
        public Value rangeFromPoint (Value point);
        /**
         * enlarges the given range by the other one
         * @param range the range to be enlarged
         * @param other the second range to enlarge
         * the first range to
         * @return the enlarged range
         */
        public Value enlargeRange (Value range, Value other);
        /**
         * enlarges the given range by a point
         * @param range the range to be enlarged
         * @param point the input value to fit the dimensions to
         * @return the enlarged range
         */
        public Value enlargeRangeByPoint (Value range, Value point);
        /**
         * checks whether a certain range contains a given point
         * @param range the range to check
         * @param point the point to check
         * @return a boolean indicator about the containment state
         */
        public boolean rangeContains (Value range, Value point);
        /**
         * checks whether the two range {@link Value}s overlap
         * @param first the first range {@link Value} to be checked
```

```
36          * for overlap
37          * @param second the second range {@link Value} to be
38          * checked for overlap
39          * @return the overlap state as boolean indicator
40          */
41         public boolean rangeOverlaps (Value first, Value second);
42         /**
43          * checks the area of enlargement between a range and a point
44          * @param range the range to be checked
45          * @param point the point to be taken as test objective
46          * @return the area of enlargement
47          */
48         public double rangeEnlargementTestPoint (Value range,
49             Value point);
50         /**
51          * tests the enlargement of one range to another one
52          * returning the area of enlargement
53          * @param range the first range to be tested
54          * @param other the second range under test
55          * @return the area of enlargement
56          */
57         public double rangeEnlargementTest (Value range, Value other);
58         /**
59          * calculates the margin of the given range
60          * @param range the range to calculate the margin for
61          * @return the margin value in double representation
62          */
63         public double margin (Value range);
64         /**
65          * calculates the area covered by the range
66          * @param range the range to calculate the coverage area for
67          * @return the coverage area of the given range
68          */
69         public double area (Value range);
70         /**
71          * retrieves the overlapping range of the first and
72          * the second one (empty range, if none)
73          * @param first the first of the two ranges
74          * @param second the second of the two ranges
75          * @return the overlapping range of the first and the
76          * second one
77          */
78         public Value overlappingRange(Value first, Value second);
79 }
```

Listing 3.10: R-Tree Operator Class Definition

The R-Tree operator class definition can be seen in listing 3.10. It contains the basic definitions which are required by the R-Tree to manage the data, properly. It supplies functions for checking the overlap or containment of ranges or points, respectively. Further functions are included to determine, e.g., the area of a range or its margin. Each of the particular functions is required by most implementations of distribution functions. Some of them represent predicates using a boolean outcome whereas others also create or calculate values like the overlapping range or the area of a given range object. These functions are mainly important for splitting or choosing a proper subtree in order to get the distribution of items inside the R-Tree working properly based on the constraints specified by the respective algorithm.

The basic interface may be implemented by data type specific realisations to promote the concrete functions on the given data types. Therefore, the most important property for all supported types is that the base types are "point-like" and the inner derived representations must behave "range-like". It is solely important that the present realisations support the given set of functions for the R-Tree to operate properly.

3.2.3.2.2 R-Tree State Similar to the B-Tree state interface (see 3.2.3.1.2), the R-Tree also contains a state interface making it more adoptable towards concrete utilisations. This means that the R-Tree may easily be used in a stand-alone environment. Yet, there is the possibility of embedding it inside other index structures by supplying a proper state implementation. This is done with respect to the hybrid index. There is a standalone version with a proper implementation of the R-Tree state interface. The hybrid index uses the R-Tree in another application specific adoption. Therefore, the main methods which must be modified include:

1. `setRoot` for setting a new root node
2. `getRoot` complementary function for `setRoot`
3. `getAdditionalInformation` providing access to additional information
4. `getAdditionalFunctions` giving access to additional functions
5. `getSplit` providing access to the split function implementation
6. `getChooseSubtree` same functionality as `getSplit`, however focused towards the algorithm to retrieve a subtree where to place a particular item

It can be seen easily that the first four functions are basically the same as those used inside the B-Tree implementation. Besides these functions, two get functions exist which give access to R-Tree specific implementation variant functions. In there, the concrete implementations for choosing a proper subtree where to place a new item as well as the split functionality which allows to split an overfull node in two is provided. Besides these basic definitions, the R-Tree also gives access to additional functions which are utilised intensively here, in order to reorganise node structures after a particular change has happened. Most of the functions are mainly required when a leaf node is modified.

```java
package de.fhhof.iisys.index.h2.rtree.util;

/**
 * interface for additional functions which are used
 * in the hybrid part of the index mainly
 * @author ckropf2
 *
 */
public interface AdditionalFunctions {
        /**
         * distributes given items after a split has occurred
         * @param origNode the {@link RTreeNode} which
         * serves as the one originally created
         * @param orig the original {@link SearchRow} that
         * was in the index before
         * @param newNode the newly created {@link RTreeNode}
         * to be modified
         * @param newOne the new {@link SearchRow} that
         * resulted from a split
         */
        public void distributeAfterSplit (RTreeNode origNode,
                SearchRow orig, RTreeNode newNode, SearchRow newOne);
        /**
         * adjusts the secondary associated structure
         * from the subtree in case of a root split
         * @param row the {@link SearchRow} to handle
         */
        public void adjustSecondaryStructure (SearchRow row);
        /**
         * gets notified when a node has been moved
         * this is useful when secondary structures might
         * have to be adjusted
         * @param node the node which has been moved
         * @param newPos the new position of the node
         */
        public void nodeMoved(RTreeNode node, int newPos);
}
```

Listing 3.11: Additional Functions Interface Definition Used by the R-tree

An overview about the methods supplied by the additional functions interface can be seen in listing 3.11. This interface basically has three methods. As it is not known by the R-Tree, how to modify the respective and potentially present secondary structure such as the bitlist in this case, it is important that there are functionalities given to achieve their proper adjustment. The function `distributeAfterSplit` reorganises the secondary structure after a split which means that in case of, e.g., the bitlist implementation it gets the new distributions for the bitlists in the respective two elements correctly. Additionally, adjustments to the elements are made. This is especially valuable in leaf nodes. They point to secondary inverted index structures. Therefore, it must be clearly defined which R-Tree leaf element is associated with which secondary inverted index structure and vice versa. This function reorganises these relations in addition to adjusting the secondary structure. Based on the bits set on a particular reorganisation operation, new bits may have been introduced or deleted. For this reason, the function `adjustSecondaryStructure` again fulfils the task of getting all bits right. In other kinds of implementations, the details may obviously vary.
The last function `nodeMoved` is specific for the implementation of the H2 database. If the (last) session to a certain database is closed, the internal structure may be reorganised. This is done

in order to keep the resulting file size small. During normal database operations, pages may be added or removed. If a page is removed somewhere inside a data range, there is a certain likelihood to generate "sparse" files which means that empty space is created between multiple data pages of the same size. To refill this empty and free space, the H2 database has a functionality to "compact" the database files on closing. The process may be forced by the use of the following SQL statement:

```
1 SHUTDOWN COMPACT;
```

Listing 3.12: Compact Statement for the H2 Database

The statement given in listing 3.12 executes the compacting of the database file and closes all remaining connections to the respective file. In standard operations, a mechanism exists which tries to reorganise the database pages on closing. Therefore, at most 200 ms of time are used (at least one move of a page is executed). In order to reorganise the moved pages, properly, manual action must be taken. That means that each page which is moved gets a request to produce a copy of itself using the new positional information. The page to be created is already allocated. Thus, only a copy must be instantiated containing the exact state of the page to be moved. This process leads to the creation of the new page whose references must be updated afterwards. So, e.g. in the case of a tree structure, the references to the parent, children as well as potentially siblings must be updated in order to keep the tree structure usable, afterwards. The adoptions which must be made inside a hybrid index storage may be more complex than just updating references to parental, children or sibling nodes. The standard operations executed on an R-Tree or a B-Tree, respectively, are carried out as is. That indicates that the "normal" node movement operations are executed. Hence, a copy of the currently inspected node is instantiated and the references to the node and from the node are adopted. This, basically, refers to the children, parent and sibling nodes (if present). Yet, regarding the structure of the hybrid index, further interdependencies between the respectively inspected storage structures exist. Hence, one important place to be inspected is, e.g., the link from the hybrid R-Tree leaf elements to the particular secondary inverted index structure. If a node is moved, the reference page for the secondary inverted index structure must be adjusted as well in the case of a leaf node. The same holds for, e.g., the position information of a root node of one secondary inverted index structure. If the root node of such a structure is moved, the entry in the respectively referenced R-Tree leaf element must be adjusted, too, in order to reorganise the entire index structure properly.
Besides the actual node movement, the R-Tree state also contains information about the particular approach to find a valid node to place an item (`getChooseSubtree()`) as well as the split mechanism (`getSplit()`). As already mentioned, the information about the algorithm to be used is passed as initialisation argument to the index structure. Having parsed the information from the initialisation string, the particular implementations are stored inside the R-Tree state. Each algorithm of the R-Tree has access to these pieces of information and thus the particular algorithms may be applied from anywhere inside the R-Tree algorithms.

3.2.4 Summary on Initial Implementation

This section gives an overview about the initial implementation of the hybrid index structure. The information delivered forms the basics for the further optimisation steps. The interconnections between the respective storage structures as well as the possibility of including customised indexing approaches are described in this section.
Some information from this chapter will be used in further chapters, especially during the description of the actual optimisation iterations. A lot of information given here is subject of change as it will be noticed that some of the approaches, described here, tend to be inefficient

for the use of hybrid index structures. During the particular optimisation iterations, there always is a focus towards the generalisation of the respective approaches, too. The developed optimizations should not only be applied to this kind of hybrid index structures but include a certain level of generalisation as well as adaptability regarding other hybrid indexing approaches. This can be validated, too, because further hybrid index structures exploring other kinds of hybrid data spaces are developed simultaneously which can be used to cross-validate the given results.

4 A GENERALISED THEORETICAL HYBRID INDEX STRUCTURE MODEL

Separated in two parts, this chapter outlines a general model for the hybrid index structure used in this thesis. It may also be applied to similar access methods which behave in a way comparable to the one used here. The first part describes a general model including the possibilities and limitations of hybrid index structures. It concludes by providing a sufficiently efficient worst case time complexity measure for hybrid access methods. The second part depicts storage mechanisms in realistic database systems leading to an analysis of the size relations inside the access method and results in the analytical derivation of the corpus dependent limit "*HLimit*" which resembles a very important parameter for the setup of the hybrid access method.
In general, this section answers research question 3 which is presented in subsection 1.2.1.3. From the view of the design science methodology, it tries to contribute a model composed of constructs and conceptualisations as a final outcome at the end of the chapter. As, currently, no theoretical definitions for the application of hybrid access structures are given, the derivation of the model is motivated by the need of a mathematical description of hybrid access structures. Although Hellerstein et al. [45] present a model of indexability, this is not sufficient for the task of describing the hybrid access method. In general, the given model works with workloads which are more suited to represent an average case and thus may not solely be used for the derivation of worst case time and space complexities. The target of the given analysis is, on the one hand, to model the worst case complexities of access methods and, on the other, derive a formal model as well as the description of the respective sizes and limitations of access structures. This section uses basic theories to produce a final model as the outcome which is related to a research contribution (see e.g. 1.2.2.1.4). Besides, the results are also presented in research articles [41, 62] for the purpose of research communication (see 1.2.2.1.7).

4.1 HYBRID INDEX STRUCTURE MODEL DERIVATION

This part of the work refers to the derivation of a general model approach facilitating the analysis of hybrid access methods in realistic database scenarios. The main results of the model generation are outlined in [41]. An adoption of the contents of [41] is given here whereas the equations, lemmas, theorems, definitions and their proofs are taken nearly literally.

The following analysis will show that storing normalised or single-valued data combined with non-normalised or multi-valued ones inside one single hybrid access structure may lead to a logarithmic search time and a linear space complexity. This can be guaranteed if there is a primary structure storing the data from the single-valued column augmented with facts from the multi-valued column, e.g., by using bitlists. The time complexity of $\mathcal{O}(\log(n) \cdot m)$ is ensured for primary tree structures storing the single-valued data which themselves ensure a logarithmic time complexity.

4.1.1 Basic Definitions and Symbols

A comprehensive list of symbols used in the further analysis is given in table 4.1.

Symbol	Meaning
E	set of entries stored in a table
e	element of set E
V	set of values represented in set E
p_i	projection function to retrieve the values $\in V$
q_i	intersections of sets of values in one column
$\mathcal{A}$	access (index) structure, represented as set
N	group inside $\mathcal{A}$
v	representative value for a group N taken from set V
C_i	search criterion for column i
$time(C_i, \mathcal{A})$	*time* function representing the number of items to be loaded from access structure $\mathcal{A}$ for criterion C_i
$space(\mathcal{A})$	size of all groups contained in access structure $\mathcal{A}$
$visit(C_i, \mathcal{A})$	function representing the groups N visited for search criterion C_i in access method $\mathcal{A}$
$result(C_i, \mathcal{A})$	function providing all results stored in access structure $\mathcal{A}$ satisfying criterion C_i

Table 4.1: Symbols Used in this Analysis

The set of entries for a database table is denoted as E and individual entries are marked as e:

$$E = \{e_1, \ldots, e_n\} \tag{4.1}$$

For the simplification of the notation it is assumed that only a single set of values V exists for all columns, e.g. the union of all domains. Projection functions may be used to retrieve the k values for individual columns. The projection functions are labelled as p_i referring to column i and may also be applied to sets of entries.

$$\begin{aligned} &p_i : E \rightarrow 2^V \text{ with } i = 1, \ldots, k \\ &p_i(\{e_1, \ldots, e_j\}) = p_i(e_1) \cup \ldots \cup p_i(e_j) \end{aligned} \tag{4.2}$$

Generally, normalised columns may not contain more than one value per entry. This primarily comes from the first normal form (see e.g. [19]) which requires atomicity of values and indicates that no composed values are allowed in normalised columns.

$$\forall e \in E : |p_i(e)| \leq 1 \tag{4.3}$$

Later in this analysis, q_i is used to indicate intersections between the projection function results:

$$q_i(\{e_1, \ldots, e_j\}) = p_i(e_1) \cap \ldots \cap p_i(e_j) \tag{4.4}$$

An important concept of a lot of access structures $\mathcal{A}$ is to assign entries to p different groups. These groups need not be disjoint.

$$\mathcal{A} = \{N_1, \ldots, N_p\} \text{ with } N_1, \ldots, N_p \subseteq E \tag{4.5}$$

Based on the distribution of the individual values, it is not necessary to visit all groups during a search. Usually, one representative value v is chosen to represent an identifier for each group N:

$$\forall N \in \mathcal{A} \exists v \in V : v \in q_i(N) \tag{4.6}$$

Another typical requirement for an index structure is that each group N contains every entry e with the respective value $v \in q_i(N)$:

$$\forall v \in V, e \in E, N \in \mathcal{A} : v \in p_i(e) \wedge v \in q_i(N) \Rightarrow e \in N \tag{4.7}$$

These basic definitions are sufficient to define an inverted index or the groups inside the leaf level of a tree structure where $\forall e \in E : |p_i(e)| \leq 1$ holds . It is important to note that conditions 4.6 and 4.7 ensure disjointness between the groups of access structures for normalised columns.

4.1.2 Complexity of Queries Addressing Single Columns

Due to the definitions above, bounds for time and space complexity may be derived. Thus, it is required to define the metrics of search conditions $C_i \subseteq V$ for column i. The search condition may either consist of a single value or a set of alternative values, like ranges. Entries containing at least one value of the search condition ($p_i(e) \wedge C_i \neq \emptyset$) are included in the final result. Therefore, groups must only be visited if at least one entry of the group is valid for the search condition:

$$visit(C_i, \mathcal{A}) = \{N | N \in \mathcal{A} \wedge (p_i(N) \cap C_i) \neq \emptyset\} \tag{4.8}$$

The result set is composed of the union of all groups visited during the search procedure:

Lemma 1. *Let $C_i = \{v_1, \ldots, v_p\}$ be a simple search condition and $\mathcal{A}$ be an access structure. Then the following function "result" provides all entries which satisfy C_i:*

$$result(C_i, \mathcal{A}) = \bigcup_{N \in visit(C_i, \mathcal{A})} N$$

Proof. With condition 4.7, a single group N exists in the access structure for every value v from C_i which provides all entries containing the value v for the given column. In addition, condition 4.6 ensures that all entries from this group N include the specified value v. Since $visit(C_i, \mathcal{A})$ contains the corresponding set for every value from C_i, the set result comprises exactly all entries satisfying the simple search condition. □

Complexity measures may be derived based on these definitions.

$$time(C_i, \mathcal{A}) = \sum_{N \in visit(C_i, \mathcal{A})} |N| \tag{4.9}$$

This function defines the number of entries in the respective access structure $\mathcal{A}$ required to be visited during a search. *time* thus may be used as measurement for the search time complexity. The second complexity measure of interest is the spatial extent of the access method which is simply the sum of the cardinalities of all groups:

$$space(\mathcal{A}) = \sum_{N \in \mathcal{A}} |N| \tag{4.10}$$

Thus, the search time of an access structure can be described by the following lemma.

Lemma 2. *Let $C_i = \{v_1, \ldots, v_p\}$ be a simple search condition and $\mathcal{A}$ be an access structure. Then the search time is limited as follows:*

$$time(C_i, \mathcal{A}) \leq p \cdot |result(C_i, \mathcal{A})| \tag{4.11}$$

Proof. With the definition of the function *time*:

$$time(C_i, \mathcal{A}) = \sum_{N \in visit(C_i, \mathcal{A})} |N|$$

With $\{N_1, \ldots N_p\} = visit(C_i, \mathcal{A})$:

$$time(C_i, \mathcal{A}) = |N_1| + \ldots + |N_p|$$

With $N_i \subseteq N_1 \cup \ldots \cup N_p$ for $i = 1, \ldots p$:

$$\begin{aligned} time(C_i, \mathcal{A}) &\leq p \cdot |N_1 \cup \ldots \cup N_p| \\ &\leq p \cdot \left| \bigcup_{N \in visit(C_i, \mathcal{A})} N \right| \end{aligned}$$

With Lemma 1:

$$time(C_i, \mathcal{A}) \leq p \cdot |result(C_i, \mathcal{A})|$$

□

This lemma shows that the number of entries $|E|$ in the respective table does not affect the search time at all.

Lemma 3. *Let $E = \{e_1, \ldots, e_n\}$ be a set of entries, $\mathcal{A}_i$ an access structure for column i and $avg_i(\mathcal{A}_i)$ the average number of values in this column i:*

$$avg_i(\mathcal{A}_i) = \frac{|p_i(e_1)| + \ldots + |p_i(e_n)|}{n} \tag{4.12}$$

Then the space required by the access structure for column i is limited by the following expression:

$$space(\mathcal{A}_i) \leq n \cdot avg_i(\mathcal{A}_i) \tag{4.13}$$

Proof. With conditions 4.6 and 4.7 every entry e occurs in not more than $|p_i(e)|$ groups. As a consequence, the summarised number of entries in the access structure $\mathcal{A}_i$ is limited by the summarised number of values in column i of all entries:

$$space(\mathcal{A}_i) \leq |p_i(e_1)| + \ldots + |p_i(e_n)|$$

This expression can be rewritten as follows:

$$space(\mathcal{A}_i) \leq n \cdot \frac{|p_i(e_1)| + \ldots + |p_i(e_n)|}{n} \leq n \cdot avg_i(\mathcal{A}_i)$$

□

This lemma describes the space of an access structure. It states that an access structure grows linearly with the number of entries inside a table, if the average number of values contributed per entry may be limited by a constant number for the respective column i.

4.1.3 Complexity of Queries Addressing Multiple Columns

Hybrid access structures generally try to integrate values from multiple columns into one single access structure. Two possibilities are currently state of the art. On the one hand, the combination of multiple into one single values is built by using combination techniques. This enables the parallel processing of both dimensions (columns). On the other, multiple access structures may be used in sequence by separating the columns and applying one specialised structure for each dimension.
In this analysis, the idea of combining multiple access structures is taken as a basis. First, pairs of access structures are combined by building the cross product of all sets. Next, the basic measures from subsection 4.1.2 are extended for multiple access structures. The analysis may be expanded by adding further access structures to the already present one(s).

Definition 1. *Let $\mathcal{A}_i$ and $\mathcal{A}_j$ be two access structures for the columns i and j. Then the combined access structure $\mathcal{A}_{ij}$ is defined as follows:*

$$\mathcal{A}_{ij} = \{N_i \cap N_j | N_i \in \mathcal{A}_i \wedge N_j \in \mathcal{A}_j\} \tag{4.14}$$

The basic functions *visit* and *time* must be extended for the cross product of access structures and multiple search criteria.

$$\begin{aligned} visit(C_i, C_j, \mathcal{A}_{ij}) = &\quad \{N | N \in \mathcal{A}_{ij} \wedge (p_i(N) \cap C_i) \neq \emptyset \wedge (p_j(N) \cap C_j) \neq \emptyset\} \\ time(C_i, C_j, \mathcal{A}_{ij}) = &\quad \sum_{N \in visit(C_i, C_j, \mathcal{A}_{ij})} |N| \end{aligned} \tag{4.15}$$

The function *visit* applied to a combined access structure $\mathcal{A}_{ij}$ which was derived from two access structures $\mathcal{A}_i$ and $\mathcal{A}_j$ provides exactly the cross product of the groups which need to be visited for the access structures $\mathcal{A}_i$ and $\mathcal{A}_j$. Note that the symbol $\oplus$ is used in this context its meaning being $A \oplus B = \{a \cap b | a \in A \wedge b \in B\}$. The elements of the respective groups A and B are groups themselves:

Lemma 4. *Let $\mathcal{A}_i$ and $\mathcal{A}_j$ be two access structures and $\mathcal{A}_{ij}$ be the combination of these access structures. Then for every pair of search conditions C_i and C_j the following property holds:*

$$visit(C_i, C_j, \mathcal{A}_{ij}) = visit(C_i, \mathcal{A}_i) \oplus visit(C_j, \mathcal{A}_j) \tag{4.16}$$

Two directions are proven. Every group N_{ij} from $visit(C_i, C_j, \mathcal{A}_{ij})$ has been generated by the intersection of two groups N_i from $\mathcal{A}_i$ and N_j from $\mathcal{A}_j$. Since entries in N_{ij} exist which satisfy both conditions, they must also belong to the groups N_i and N_j. As a consequence, N_i has to be in $visit(C_i, \mathcal{A}_i)$ and N_j has to be in $visit(C_j, \mathcal{A}_j)$ and it can be concluded that $N_{ij} = N_i \cap N_j$ is in

$$visit(C_i, \mathcal{A}_i) \oplus visit(C_j, \mathcal{A}_j)$$

In a similar way it can be proven that every element $N_i \cap N_j$ of $visit(C_i, \mathcal{A}_i) \oplus visit(C_j, \mathcal{A}_j)$ is also an element of $visit(C_i, C_j, \mathcal{A}_{ij})$. With lemma 1 every entry in N_i fulfils the search condition C_i and every entry in N_j satisfies the search condition C_j. This means that all entries of the intersection of these groups satisfy both conditions and the intersection is a member of $visit(C_i, C_j, \mathcal{A}_{ij})$ concluding the proof.

Lemma 1 can be extended for combined access structures:

Lemma 5. *Let C_i and C_j be two simple search conditions and $\mathcal{A}_{ij}$ be a combined access structure. Then the following function "result" provides all entries which satisfy both search conditions:*

$$result(C_i, C_j, \mathcal{A}_{ij}) = \bigcup_{N \in visit(C_i, C_j, \mathcal{A}_{ij})} N \tag{4.17}$$

Proof. The result for a query with the search conditions C_i and C_j is the intersection of the result sets for the individual conditions:

$$result(C_i, C_j, \mathcal{A}_{ij}) = result(C_i, \mathcal{A}_i) \cap result(C_j, \mathcal{A}_j)$$

With lemma 1 this results in:

$$\begin{aligned} result(C_i, C_j, \mathcal{A}_{ij}) &= \left(\bigcup_{N \in visit(C_i, \mathcal{A}_i)} N \right) \cap \left(\bigcup_{N \in visit(C_j, \mathcal{A}_j)} N \right) \\ &= \bigcup_{N \in visit(C_i, \mathcal{A}_i) \oplus visit(C_j, \mathcal{A}_j)} N \end{aligned}$$

With lemma 4:

$$result(C_i, C_j, \mathcal{A}_{ij}) = \bigcup_{N \in visit(C_i, C_j, \mathcal{A}_{ij})} N$$

□

This leads to the deduction of an upper bound for the search time complexity of such a structure:

Lemma 6. *Let $\mathcal{A}_{ij}$ be an access structure derived from the two access structures $\mathcal{A}_i$ and $\mathcal{A}_j$ and $C_i = \{v_{i1}, \ldots, v_{ip}\}$ and $C_j = \{v_{j1}, \ldots, v_{jq}\}$ be two search conditions. Then the search time is limited by the following expression:*

$$time(C_i, C_j, \mathcal{A}_{ij}) \leq p \cdot q \cdot |result(C_i, C_j, \mathcal{A}_{ij})| \tag{4.18}$$

Proof. With the definition of the function *time*:

$$time(C_i, C_j, \mathcal{A}_{ij}) = \sum_{N \in visit(C_i, C_j, \mathcal{A}_{ij})} |N| = \sum_{N \in visit(C_i, \mathcal{A}_i) \oplus visit(C_j, \mathcal{A}_j)} |N|$$

With $\{N_{i1}, \ldots, N_{ip}\} = visit(C_i, \mathcal{A}_i)$ and $\{N_{j1}, \ldots, N_{jq}\} = visit(C_j, \mathcal{A}_j)$, this expression can be rewritten as follows:

$$\begin{aligned} time(C_i, C_j, \mathcal{A}_{ij}) = \; & |N_{i1} \cap N_{j1}| + \; \ldots \; + |N_{ip} \cap N_{j1}| + \\ & \qquad \vdots \qquad \ddots \\ + \; & |N_{i1} \cap N_{jq}| + \; \ldots \; + |N_{ip} \cap N_{jq}| \end{aligned}$$

Since each expression of the form $N_{ix} \cap N_{jy}$ is a subset of $result(C_i, C_j, \mathcal{A}_{ij})$ the following upper bound can be derived:

$$\begin{aligned} time(C_i, C_j, \mathcal{A}_{ij}) \leq \; & |result(C_i, C_j, \mathcal{A}_{ij})| + \; \ldots \; + |result(C_i, C_j, \mathcal{A}_{ij})| + \\ & \qquad \vdots \qquad \ddots \\ + \; & |result(C_i, C_j, \mathcal{A}_{ij})| + \; \ldots \; + |result(C_i, C_j, \mathcal{A}_{ij})| \end{aligned}$$

This formula with p columns and q rows can be simplified as:

$$time(C_i, C_j, \mathcal{A}_{ij}) \leq p \cdot q \cdot |result(C_i, C_j, \mathcal{A}_{ij})|$$

□

This leads to the same conclusion as the analysis of the time restrictions for single columns: The search time only depends on the size of the result set and not at all on the number of entries in the user table. Yet, the definition of combined access structures might causes unacceptable space requirements as building the cross product for all groups potentially evolves high demands of space.

Lemma 7. *Let k be the k-th value in a column, $i_k = avg_i + \overline{i_k}$ and $j_k = avg_j + \overline{j_k}$. Then the space of a combined access structure space($\mathcal{A}_{ij}$) may be expressed as:*

$$\begin{aligned} space(\mathcal{A}_{ij}) &= n \cdot avg_i \cdot avg_j + \sum_{k=1}^{n} (\overline{i_k} \cdot \overline{j_k}) \\ space(\mathcal{A}_{ij}) &= n \cdot avg_i \cdot avg_j + \sum_{k=1}^{n} (avg_i - i_k) \cdot (avg_j - j_k) \end{aligned}$$

Proof. Based on the combination of columns, the space necessary for each combined value is the product of the particular affected values i_k and j_k. The space for the k-th value can be expressed as:

$$(avg_i + \overline{i_k}) \cdot (avg_j + \overline{j_k}) = avg_i \cdot avg_j + avg_i \cdot \overline{j_k} + avg_j \cdot \overline{i_k} + \overline{i_k} \cdot \overline{j_k}$$

To obtain the total space, this must be applied to all n rows in a table:

$$\begin{array}{rllll} space(\mathcal{A}_{ij}) = & avg_i \cdot avg_j + & avg_i \cdot \overline{j_1} + & avg_j \cdot \overline{i_1} + & \overline{i_1} \cdot \overline{j_1} \; + \\ + & avg_i \cdot avg_j + & avg_i \cdot \overline{j_2} + & avg_j \cdot \overline{i_2} + & \overline{i_2} \cdot \overline{j_2} \; + \\ + & avg_i \cdot avg_j + & avg_i \cdot \overline{j_3} + & avg_j \cdot \overline{i_3} + & \overline{i_3} \cdot \overline{j_3} \; + \\ + & & \dots & & \; + \\ + & avg_i \cdot avg_j + & avg_i \cdot \overline{j_n} + & avg_j \cdot \overline{i_n} + & \overline{i_n} \cdot \overline{j_n} \end{array}$$

The sum of all terms of the form $avg_i \cdot \overline{j_k}$ and $avg_j \cdot \overline{i_k}$ is zero:

$$\begin{aligned} avg_i &= \tfrac{1}{n} \sum_{k=1}^{n} i_k \\ &= \tfrac{1}{n} \sum_{k=1}^{n} (avg_i + \overline{i_k}) \\ &= \tfrac{avg_i + \overline{i_1} + avg_i + \overline{i_2} + avg_i + \overline{i_3} + \ldots + avg_i + \overline{i_n}}{n} \\ &= \tfrac{n \cdot avg_i}{n} + \tfrac{\overline{i_1} + \overline{i_2} + \overline{i_3} + \ldots + \overline{i_n}}{n} \\ \Rightarrow avg_i &= avg_i + \tfrac{\overline{i_1} + \overline{i_2} + \overline{i_3} + \ldots + \overline{i_n}}{n} \\ \Rightarrow \quad & \tfrac{\overline{i_1} + \overline{i_2} + \overline{i_3} + \ldots + \overline{i_n}}{n} \stackrel{!}{=} 0 \end{aligned}$$

Therefore, the remaining terms are:

$$\begin{aligned} space(\mathcal{A}_{ij}) &= \sum_{k=1}^{n} (avg_i \cdot avg_j) \cdot \sum_{k=1}^{n} (\overline{i_k} \cdot \overline{j_k}) \\ &= n \cdot avg_i \cdot avg_j + \sum_{k=1}^{n} (\overline{i_k} \cdot \overline{j_k}) \end{aligned}$$

□

This lemma shows the required space for a combined access structure $\mathcal{A}_{ij}$. The product of $n \cdot avg_i \cdot avg_j$ already yields a very large space requirement. Yet, if the average number of values in both columns is sufficiently low and the differences between the averages in the second part may be accepted as low, too this could be feasible in some cases. However, this may not be assumed in the general case. Especially, if the sum of the products of the differences of the averages ($\sum_{k=1}^{n} (\overline{i_k} \cdot \overline{j_k})$) becomes positively large, the additional addend also leads to very large space requirements. Thus, this lemma may lead to very high space demands in the general case and may only show moderate ones in some, potentially rare, cases.

4.1.4 Tree Structure for Navigating to Sets of Access Structures

As already mentioned, the groups of not necessarily distinct entries are often accessed via tree structures. Generally, the representative value v is applied in these tree structures to separate subspaces to navigate to the groups. The groups themselves are organised below the leaf node. The tree structures are employed as directory for the navigation to the sets. As the intended use for this access structure is a disk-resident database system, only tree structures for secondary memory are taken into account. One commonly adopted and worst-case efficient storage structure is the B(+)-Tree which ensures a search time complexity of $\mathcal{O}(\log(n) + m)$ for a structure with n entries and a query producing m results. Unfortunately, the R-Tree cannot guarantee this time complexity in any case. However, there are tree packing methods also ensuring a logarithmic time complexity for a static dataset in two dimensions (see e.g. [37]). Therefore, this analysis also applies to R-Tree based structures if they are appropriately pre-packed by using adequate pre-calculation methods. Both structures, B(+)-Tree and R-Tree, ensure a linear space complexity ($\mathcal{O}(n)$). Yet, the subsequent analysis assumes the use of B+-Trees.
Three options are available for the ensuing analysis:

1. Concatenation of multiple structures, which means that a primary structure stores values from one column followed by a secondary structure containing the data from the second column. This procedure may be extended to an arbitrary amount of columns / dimensions.
2. Alteration of the nodes on the path leading to a level- / cluster-wise change in the inspected column. That means that, e.g., the first level refers to the first column and the second level to the second column followed by the third level which, again, cares about values for the third column. This option is not considered in this analysis.
3. Augmentation of node contents of a primary structure by information of a secondary structure. This option combines values of multiple columns inside one storage structure and orders based on one of the respectively inspected column values.

As a prerequisite for this analysis, two access structures $\mathcal{A}_i = \{N_{i1}, \ldots N_{ip}\}$ and $\mathcal{A}_j = \{N_{j1}, \ldots N_{jq}\}$ for the columns i and j are taken into account. Two tree structures are used where the first one (primary tree) organises the p groups of $\mathcal{A}_i$ which form the starting point to the secondary trees storing the groups of $\mathcal{A}_j$. Obviously, the space of this combination of structures depends on the distribution of values to the groups.
First, the space complexity is inspected. This case is trivial for the access structure $\mathcal{A}_i$ as it may directly be taken from equation 4.10. The space complexity for the particular secondary trees depends on the cardinalities of the groups referred to by each group from the primary tree. The average space required per group can be derived by the following equation:

$$\frac{space(\mathcal{A}_i)}{p} = n \cdot \frac{avg_i}{p} \tag{4.19}$$

For the analysis of the total space, inspecting the growth or the current size of groups is necessary. Growths of sets or sizes in pre-defined points in time may easily be derived by the use of Heaps' law [44]. This empirical law relates the number of unique entries inside a "vocabulary" with respect to the total number of entries. It is applied to the average space per group as follows:

$$p_j(N) \approx K_j \cdot \left(n \cdot \frac{avg_i}{p}\right)^{\beta_j} \tag{4.20}$$

K_j and β_j are application specific parameters.
This basic equation describes the growth of the total number of values for column j. Each of the values from column i is the starting point of exactly one secondary tree based on the chaining of access structures. That indicates that there are p groups (and also secondary trees) in the

secondary access structure. Based on the fact that the space grows linearly with the number of values, the total space can be computed by:

$$K_j \cdot \left(n \cdot \frac{avg_i}{p} \right)^{\beta_j} \cdot p = K_j \cdot \left(n^{\beta_j} \cdot avg_i^{\beta_j} \cdot p^{-\beta_j} \right) \cdot p = K_j \cdot n^{\beta_j} \cdot avg_i^{\beta_j} \cdot p^{1-\beta_j} \tag{4.21}$$

The inspection of the sizes may consider two cases for the allocation of the columns:

1. single-valued column i and multi-valued column j
2. multi-valued column i and single-valued column j

The first option (single-valued first, item 1) leads to an average number of 1 for the term $avg_i^{\beta_j}$ because not more than one entry is added per entry in the single-valued column. In contrary, the second option (multi-valued first) can contribute a very high average value. This is, e.g., the case when texts are inspected. Each text may contribute hundreds of individual new terms per entry in the table. Besides, the number of p groups is also much greater for option 2. Due to the potentially large growth of the primary access structure for the option "multi-valued first", it is ignored further on. Yet, it must be noted that under certain circumstances, this option might also be efficiently supported regarding its space. However, this is not true in the general case. Thus, in the following, only the option "single-valued first" (item 1) will be considered. Hence, under the prerequisite that $avg_i^{\beta_j} = 1$, equation 4.21 may be rewritten as:

$$K_j \cdot n^{\beta_j} \cdot p^{1-\beta_j}$$

With a linear growth of entries in column i and $n \geq p$:

$$K_j \cdot n \tag{4.22}$$

This equation defines a linear space complexity for the concatenated access structure if column i is taken as the primary access structure.
The time complexity for this access structure is composed by the one from the primary access structure and the individual ones from the secondary structures. It supports the retrieval of m_i intermediate results from column i, each of which representing the starting point to secondary trees managing n_k entries from the respective group. A B(+)-Tree is assumed for both of the access structure parts. The secondary trees return m_{i_k} entries per tree. Additionally, it may safely be assumed that the groups of the single-valued column are disjoint. With the assumption that m is the total quantity of the final result set:

$$\log(n) + \sum_{k=1}^{m_i} (\log(n_k) + m_{i_k}) = \log(n) + \sum_{k=1}^{m_i} \log(n_k) + \sum_{k=1}^{m_i} m_{i_k} = \log(n) + \sum_{k=1}^{m_i} \log(n_k) + m \tag{4.23}$$

Compared to a conventional B(+)-Tree, this time complexity contributes the additional term $\sum_{k=1}^{m_i} \log(n_k)$ which might result in a linear time complexity. This originates from the fact that all of the secondary trees included in the result set from the primary access structure need to be searched. Unfortunately, until this moment, no information has been provided about the contents of the secondary tree. Hence, there is a certain likelihood that a large range of the secondary trees must be examined in depth without contributing one single result.
In order to avoid this potentially harmful behaviour, a solution must be found which guarantees that a certain number ε of entries may be found in a secondary tree if the search process continues to one group. Optimally, this number should not be dependent on the number n of entries. This can be ensured by augmenting the values from column i in the primary access structure with information of column j by annotating the values from column j that occur below a respective node in the access structure $\mathcal{A}_i$. The augmentation of the node elements with bitlists, which is done by the hybrid access method utilised in this work may be most helpful here.
The result of the analysis is summarised in the following theorem.

Theorem 1. *Let $\mathcal{A}_{ij}$ be a combined access structure for the single-valued column i and the multi-valued column j using concatenated B+-Trees for the navigation to the groups from $\mathcal{A}_{ij}$. The primary tree refers to column i and the secondary tree refers to column j. The references of the primary tree are augmented in such a way that a search continues beyond a reference if at least ε entries from the referenced structure satisfy the conditions for column j. Then the search time is limited by the following expression where n is the number of entries and m is the size of the result set:*

$$\mathcal{O}\left(\log(n) \cdot \frac{m}{\varepsilon} + m\right)$$

Proof. In the previous subsection it could already be proven that the time complexity of an access structure does not depend on the number of entries. Therefore, the focus is directed towards the time required to navigate through the tree structure. Based on equation 4.23 and the assumption that the search continues only beyond a reference of the primary structure if at least ε entries satisfy the search condition for column j, it can be concluded that not more than $\frac{m}{\varepsilon}$ secondary tree structures are searched.
Additionally, the number of entries in a secondary access structure is limited by n resulting in a search time complexity of:

$$\leq \log(n) + \sum_{k=1}^{\frac{m}{\varepsilon}} \log(n) + m$$

Now, this expression may be rewritten as follows:

$$\leq \log(n) + \frac{m \cdot \log(n)}{\varepsilon} + m$$

Since only the fastest growing term needs to be considered for the $\mathcal{O}$-notation, the following expression resembles the upper bound for the search time complexity (see lemma 2):

$$\mathcal{O}\left(\log(n) \cdot \frac{m}{\varepsilon} + m\right)$$

□

The most interesting fact here is that in case a logarithmic search time complexity can be achieved by the primary access structure which must be combined with some kind of markup for the secondary values, the entire search effort is logarithmic. Thus, if a spatial structure can be found supporting logarithmic retrieval behaviour like a specialised pre-packed R-Tree structure, also the entire search effort can be limited to a logarithmic behaviour.
Most of the facts, especially the equations, lemmas, theorems and definitions from this subsection, originate nearly literally from [41]. Yet, they are very important for this work as they build the foundation of the theoretic derivation of a model which is used subsequently as a basis for this thesis. One instance of an index structure using a B-Tree as primary directory storage is presented in [65]. This index structure is used to support the search process for named entities derived from texts to build a combined search possibility of identified objects in conjunction with keywords.

4.2 ANALYSIS OF STORAGE STRUCTURES AND DERIVATION OF *HLIMIT*

After the introduction of the facts for a general hybrid index structure model, further limitations are inspected, now.
Parts of this subsection are already published in an article (see [62]).

As a first step, a table of symbols is given which introduces the names of the variables deployed here. This symbol table is then applied in a description of disk oriented storage generally leading to tree structures as well as an introduction of tree based methods for an inverted index. A deduction of features for a hybrid index structure pursues the introduction of both aspects then. The main goal of this model is to describe the respective storage areas used by the hybrid index structure. Besides the set sizes and constructions, also the sizes of the storage structures applied inside the hybrid index are derived. Hence, this model forms the basis for this work and depicts, from a conceptual view, the respectively affected parts of the index structure. Certain properties which are used later in this work are already delineated here.
First, the general approach of storing entries in disk blocks and the management of pages are depicted. A central result of this analysis is that one page which is composed of multiple fields may be described as a set of elements. The next part deals with the storage of elements in balanced tree structures for non-normalised values which derives some general properties. Hereafter, the storage possibilities of non-normalised values in inverted index structures are outlined. The two kinds of storage structures are then combined to the hybrid index whose general properties regarding the sets and space limitations are also inspected.

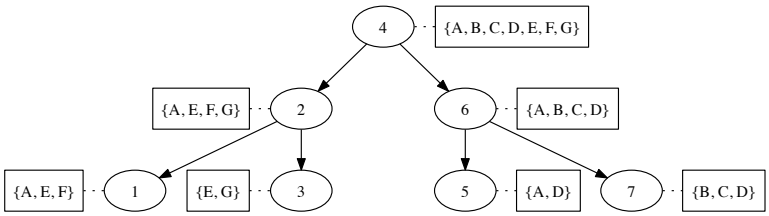


Figure 4.1: Conceptual Representation of a Hybrid Index Structure

A general survey image of hybrid index structures separating sets of non normalised values (e.g. textual components) can be seen in figure 4.1. The basic structure chosen for illustration here, is a binary tree for the normalised values distributing a set of non-normalised values annotated at the particular nodes. Afterwards, the structures involved in the construction of such an index structure in combination with a disk oriented storage approach separating the items in blocks is described.

A comprehensive list of symbols used for the description of the hybrid index structure model can be depicted from table 4.2 which is comparable to table 4.1 from subsection 4.1.

An overview of the terminology used for the description of index access methods is shown in chart 4.3. It describes frequently used terms as well as their meaning in context.

4.2.1 Block Oriented Storage

Block oriented storage approaches are often used by hard disk oriented database systems. Hard disks themselves are organised in blocks on which the respective operating systems may perform workflows by applying a file system.
Generally, a physical block is mapped to a page. This page may be extended by specific information portions. There is also free space for the real contents.
Therefore:

$$p = (h, c) \tag{4.24}$$

Here, page p represents a tuple which is comprised of header information h and contents c. Normally, header information always has the same size regarding the same types of pages. That means that this is just an additive size which is not relevant for further inspections.

Symbol	Meaning	Type
$\langle X \rangle$	storage space required by storage structure X	integer
p	page in a block oriented storage system (tuple representation)	tuple
h	header information in a page	undefined
c	page content	undefined
n	amount of entries to be stored in a page / node	integer
o	offset inside a page	integer
ce	content element stored in an item constructed by item content and a pointer	undefined
ic	item content	undefined
pt	pointer for referencing a storage area / item	pointer
e	element / entry stored inside one page	element
P	page in a block oriented storage system (set representation)	set
UT	user table storing the original tuples (set)	set
t	tuple stored inside a user table	tuple
a	attribute stored in a tuple row cell	element
$m(t)$	mapping function	function
$m^{-1}(e)$	inverse mapping function	function
ik	representative item value (referred to as item key)	element
R	set of relational / normalised entries (derived from tuple values)	set
rv	element in the set of relational / normalised entries	element
LE	set of leaf entries	set
sk	symbolic key	undefined
N	set of non-relational / -normalised content elements	set
nv	entries in the set of non-relational / -normalised content elements	element
$inv(n)$	invert function for a certain n mapping the element to an inverted list	function
L	posting list for a particular item	list / set
II	inverted index	set
$v(q)$	vocabulary size of individual values when there are total q values	function
K	constant pre-factor to be used in Heaps' Law	real
β	constant exponent to be used in Heaps' Law	real
$z(nv)$	Zipf's Law function	function
z	Zipf's Law value	real
$rank(nv)$	rank function to get the rank for a given non-normalised entry	function
$limit$	artificial upper bound for items in mixed storage of inverted index	integer
m	size of the result set for queries	integer
s	number of query objects (e.g. non-normalised)	integer
rp	relational part of the object to be indexed	element
np	non-relational part of the object to be indexed	element
$w(t)$/ $w^{-1}(nv)$	projection function from tuple to set of non-normalised items and inverse function	function
$p(t)$/ $p^{-1}(rv)$	projection function from tuple to set of normalised items and inverse function	function
$HLimit$	artificial upper bound value to divide the sets into high frequently and low frequently occurring items	integer
bi	a bit index assigned by the hybrid index	integer

Table 4.2: Symbols Used for the Description of Index Access Methods

Term	Meaning
element	element stored inside a page or node, respectively
value	value entry as representative element of a set
entry	synonym for element, mainly used for the description of the storage of elements inside disk blocks
item	content of one entry composed by item content and pointer (for database pages)

Table 4.3: Terms Used for the Description of Index Structures

A database system working block oriented uses the same basic information as seen in equation 4.24. Header information therefore gives hints on properties of the respective pages. In case of a tree, this header contains, e.g., information about a parent node, the amount of entries stored inside the page or about sibling nodes. Generally, as this header information is constant for pages of same type, this storage area may be ignored for the following analysis.
Each page has a pre-defined storage size which is set not later than the creation time of a database. In some cases this size may already be set at the compile time of the database.
The amount of storage applied by a certain structure is denoted as $\langle X \rangle$. X is a symbol which requires the storage.
The amount of storage used by a certain page p may then be denoted as:

$$\begin{aligned} \langle p \rangle &= \langle h \rangle + \langle c \rangle \\ \langle h \rangle &= const. \end{aligned} \tag{4.25}$$

In equation 4.25 $\langle h \rangle$ is a constant factor which can be neglected in the following inspection. The only variable size content of the page can be depicted by $\langle c \rangle$.

Figure 4.2: Filling Strategy of a (Database) Page

Image 4.2 shows a typical distribution of storage areas inside a (database) page and the method of filling it. This storage approach is commonly used in database systems and inspired by the N-ary Storage Model (NSM, see, e.g. [82, pp. 327 – 333]). The header area comes first (sometimes also last). The offsets which map the positions of the respective items inside a page follow. The items are filled into the page from the end to the front until the free space in the centre is filled completely, which means that one item would overwrite an offset.
Therefore, the remaining content of the page may be described as:

$$\begin{aligned} \langle c \rangle &= \sum_{i=1}^{n} (\langle o \rangle + \langle ce_i \rangle) \\ &= n \cdot \langle o \rangle + \sum_{i=1}^{n} \langle ce_i \rangle \end{aligned} \tag{4.26}$$

As can be seen in equation 4.26, it is already obvious that the offsets consist of a constant factor solely depending on the number n of entries or content elements (ce) stored inside the page. The remaining, possibly dynamically sized, area to be evaluated is the size of one content element. Typically, in database systems, entries stored in a page are comprised of the actual content element and a pointer or reference to another storage structure. These pointers include references to related pages, e.g. a child page in a tree structure, or represent back pointers to elements, e.g. a reference to the row referenced by an index. This leads to the following representation:

$$ce = (ic, pt) \tag{4.27}$$

ic is the item content of one entry stored inside the page and pt represents the pointer reference. Generally, these pointers are also of fixed size ($\langle pt \rangle = const.$), which means that the

only dynamic variable inside the system to estimate the size for remains the real item content *ic*. Referring to equation 4.26, it follows from the above:

$$\langle c \rangle = n \cdot \langle o \rangle + n \cdot \langle pt \rangle + \sum_{i=1}^{n} \langle ic_i \rangle \tag{4.28}$$

Therefore:

$$\langle c \rangle = n \cdot (\langle o \rangle + \langle pt \rangle) + \sum_{i=1}^{n} \langle ic_i \rangle \tag{4.29}$$

Totalling (see equation 4.25):

$$\langle p \rangle = \langle h \rangle + n \cdot (\langle o \rangle + \langle pt \rangle) + \sum_{i=1}^{n} \langle ic_i \rangle \tag{4.30}$$

$\langle h \rangle$, $\langle o \rangle$ and $\langle pt \rangle$ are constant factors and the amount of entries n which can be stored inside a page simply depends on the particular sizes of items ic_i.
Regarding the sizes of items ic_i it can be stated that there are two cases:

1. fixed size information (e.g. numbers, ...)
2. variable size information (e.g. strings, ...)

Case 1 represents the storage information of fixed size. This implies that also the required amount of the storage of each item content requires the same amount of space. Consequently (as $\langle ic_i \rangle = \langle ic_j \rangle \; \forall i, j \in \mathbb{N}$):

$$\begin{aligned} \langle p \rangle &= \langle h \rangle + n \cdot (\langle o \rangle + \langle pt \rangle) + \sum_{i=1}^{n} \langle ic_1 \rangle \\ &= \langle h \rangle + n \cdot (\langle o \rangle + \langle pt \rangle) + n \cdot \langle ic_1 \rangle \\ &= \langle h \rangle + n \cdot (\langle o \rangle + \langle pt \rangle + \langle ic_1 \rangle) \end{aligned} \tag{4.31}$$

Based on equation 4.31 the amount of entries (n) which may be stored in such a page is then given as:

$$n = \frac{\langle p \rangle - \langle h \rangle}{\langle o \rangle + \langle pt \rangle + \langle ic_1 \rangle} \tag{4.32}$$

The situation becomes more complex, however, regarding case 2. Here, the size of a single item content generally depends on the size s of a base information which itself can be expressed as a fixed size information part and the amount of occurrences k of this information inside a certain item content. Hence, this item is comprised by a variable length set of independent portions of information, which is, e.g., the case when inspecting the storage of strings in pages. It implies that the amount needed for storing one item then depends on the number of individual information portions which form the item. That means:

$$\langle ic \rangle = k \cdot \langle s \rangle \tag{4.33}$$

In general, for example in the case of strings, it can be stated that the amount of individual portions of information (length of the item) follows a certain probability density distribution. Therefore, there is a certain probability density function which represents the distribution of the respective information about k. If there are strings, this function might be a standard normal distribution ($f(k) = \frac{1}{\sqrt{2\pi}} e^{\frac{-k^2}{2}}$). Each distribution has some basic parameters, like a central tendency, e.g. the arithmetic mean. To facilitate further calculations, the arithmetic mean $\overline{k}$ is now taken for the central tendency parameter.

Assuming that there is a probability distribution and a central tendency parameter, it can be stated that

$$\langle ic_i \rangle = \overline{k} \cdot \langle s \rangle, \ \forall i \in \mathbb{N} \tag{4.34}$$

This brings about the calculation of size:

$$\begin{aligned} \langle p \rangle &= \langle h \rangle + n \cdot (\langle o \rangle + \langle pt \rangle) + \sum_{i=1}^{n} (\overline{k} \cdot \langle s \rangle) \\ &= \langle h \rangle + n \cdot (\langle o \rangle + \langle pt \rangle) + n \cdot \overline{k} \cdot \langle s \rangle \\ &= \langle h \rangle + n \cdot (\langle o \rangle + \langle pt \rangle + \overline{k} \cdot \langle s \rangle) \end{aligned} \tag{4.35}$$

Thus, based on equation 4.35, the amount of entries (n) which can be stored inside one particular page can be expressed as

$$n = \frac{\langle p \rangle - \langle h \rangle}{\langle o \rangle + \langle pt \rangle + \overline{k} \cdot \langle s \rangle} \tag{4.36}$$

Equations 4.32 and 4.36 show that the amount of items to be stored inside one page can either be expressed based on one particular sample item, as in option 1, or because of a probability distribution, as in option 2.
Due to the fact that one page is generally represented as a fixed size header information and a free space and each entry (at least in the average case) requires the same amount of space, it can be stated that a particular page P can be expressed as a set of entries e_i without the loss of generality.

$$P = \{e_1, e_2, e_3, \ldots, e_n\} \tag{4.37}$$

4.2.2 Relational Index Structures

If a relational index in classical disk oriented database management systems is intended to be persistent, it is stored on disk pages. The exact storage procedure of entries on a disk is explained in section 4.2.1. Based on the argumentation given in this chapter, relying especially on equations 4.32, 4.36 and 4.24, it is evident that one particular page P can be represented as $P = \{e_1, e_2, e_3, \ldots, e_n\}$.
A relational index structure is constructed to handle tuples which are stored in database tables. The items to be indexed are identified by using some kind of relational representative value. As shown in the previous section, it may be assumed that the amount of items to be stored inside one page is constant to a certain factor n which is limited by the block / page size $\langle p \rangle$.
Here, primarily tree oriented balanced storage structures like the B-Tree or the R-Tree are inspected.

4.2.2.1 General Indexing

Regarding a specific user table UT and its contents delineated as tuples t, the table can be represented as a set of tuples

$$UT = \{t_1, t_2, t_3, \ldots, t_t\} \tag{4.38}$$

where the table stores t items. The rows themselves are constructed as tuples having multiple single attributes (column cells) of arbitrary type which is specified by the column type:

$$t = (a_1, a_2, a_3, \ldots, a_m) \tag{4.39}$$

Equation 4.39 shows one tuple t which is made of several attributes a of length m.
When it comes to indexing, (at least) one representative value is chosen to identify a particular tuple inside a table. This representative value may either be an attribute or a projection applied to an attribute or a set of attributes which form the value. Therefore, a mapping exists which

derives a representative entry value (referred to as item key *ik* in the following) from a certain tuple *t* and an inverse function mapping from the item key *ik* to all tuples containing this item key:

$$\begin{aligned} m(t) &= ik \\ m^{-1}(ik) &= \{t_1, t_2, t_3, \ldots, t_i\} \end{aligned} \tag{4.40}$$

As base for the storage, a set of normalised values R exists which consists of all possible particular values valid for the stored type:

$$R = \bigcup_{t \in UT} m(t) = \{rv_1, rv_2, rv_3, \ldots, rv_p\} \tag{4.41}$$

In general, it can be stated that the item keys *ik* are contained in the set of normalised values R ($ik \in R$). Thus, the relational index normally stores the inverse mapping function m^{-1} using the mechanics of storing a representative item key $rv \in R$ and a pointer *pt* which consequently depicts the mapping function and points back to each original tuple *t* it stands for. Thus, a search request towards one value is supported by applying an item key as query parameter which shall retrieve all elements equalling this search key. Other query types, like range searches specifying a lower and upper bound for valid elements, may be supported.

4.2.2.2 Relational Tree Storage

Index structures for relational attribute data are generally stored in tree organisation in relational database management systems. Typical examples for storing relational data in index structures are, e.g., the B-Tree or the R-Tree. These trees store the data of the nodes inside database disk pages. Hence, two different storage structures exist: leaf and inner nodes. They refer to pages where the nodes are stored in. So, P_l represents a leaf page and P_i an inner page inside a tree structure. The general item structure stored in a page is already explained in section 4.2.1. It can be assumed that one particular element stored inside a page is comprised by the item key (here *rv*) and a pointer *pt*. In the case of these two features, once again the distinction between inner and leaf storage must be made. If there is a leaf node, the pointer pt_l points back to the original tuple *t* (or a secondary storage structure). The pointers contained in inner entries pt_i point to other nodes / pages inside the tree. These might again be inner or leaf nodes. As already illustrated in section 4.2.1, the sizes of these pointers do not affect the relations in storing items on pages at all. For this reason, they are left out for further discussions. However, it is important to keep the meaning of these pointers in mind.
A leaf page stores the basic information and thus implements the mapping from the item key *rv* to the tuple *t*, which means that here the inverse function m^{-1} is implemented. Generally, in the case of relational components, the respective values are ordered or clustered based on some pre-defined criterion. Regarding, e.g., a B-Tree or its variants, the ordering normally is done ascendingly. Therefore, in one page $P = \{e_1, e_2, e_3, \ldots, e_n\}$ it is always true that $e_i < e_j,\ \forall i < j\, \forall i, j \in \mathbb{N}$.
As to tree storage a general property of balanced trees is that at least two entries are stored inside one page, except for the root node. The maximum number of entries stored in one page is limited by $\langle p \rangle$, which leads to the maximum amount *n* as described in equation 4.36 or 4.32. The sum of entries to be stored in leaf nodes of a structure is, at maximum, proportional to the amount of tuples inside the user table:

$$|LE| \leq |UT| \tag{4.42}$$

This is due to the fact that each new tuple contributes at maximum one new normalised value. Thus, the leaf nodes directly store the user entry mappings m^{-1}. For inner nodes, one representative value is derived from the child nodes which separates the search space into, optimally disjunct, subspaces which are indexed by separate subtrees. So, a symbolic key *sk* exists which is used as key for a child node indexed in a lower level. Therefore, the content element (*ce*) for leaf nodes is comprised by the item key *ik* and a pointer to the original user

table entry pt_l in leaf nodes and by the symbolic key sk and a pointer to a child page pt_i in the case of inner nodes. In a B$^+$-Tree, this symbolic key equals one of the keys of a child node which separates the entries. Generally, the first key in the child node pointed to by the respective pointer is taken as a symbolic key. However, this symbolic key can also be constructed in another way. Normally, at least in inner nodes, the symbolic key describes the contents of all elements occurring in the subtree pointed to by the element. For B$^+$-Tree like structures where the symbolic key is simply derived by taking the first (or last, respectively) element of the subtree as a separator, the size of the symbolic key is equal to the size of one item key ($\langle sk \rangle = \langle ik \rangle$). For index structures, like the R-Tree, this can also change, if the R-Tree stores points in leaf nodes and range values in inner nodes. Then the size of the symbolic key $\langle sk \rangle$ is twice as large as the item key size $\langle ik \rangle$ ($\langle sk \rangle = 2 \cdot \langle ik \rangle$) because the range consists of a lower and an upper bound in each dimension. So, in this case, the amount of elements which can be stored in inner nodes (n_i) is also smaller than the amount of entries to be stored in leaf nodes (n_l). On that condition it can generally be stated that n_i is approximately half as big as n_l which leads to another branching factor for inner nodes, in this case. The derivation of the symbolic keys is application dependent and induced by, on the one hand, the index structure itself and, on the other, by its use and application domain.
To derive the upper bound for the branching factor, these two limits (n_l and n_i) need to be regarded. The lowest level consists of (at minimum) $|P_l| = \frac{|LE|}{n_l}$ leaf nodes. The next level consists of (again at minimum) $|P_{i_d}| = \frac{|P_l|}{n_i}$ inner nodes. As the relational tree index is constructed recursively until only one (root) node remains, the subsequent parental levels are comprised by $|P_{i_{d+1}}| = \frac{|P_{i_d}|}{n_i}$. These cases are only true, if the tree is filled to the maximum extent i.e. that each node stores its maximum possible amount of entries. Assumed that $\langle ik \rangle = \langle sk \rangle$, as it is generally true for standard B-Trees and variants, the depth / height of the tree h can be described as (as $n = n_i = n_l$):

$$h = \log_n |LE| \tag{4.43}$$

The amount of leaf nodes grows linearly with the number of entries to be stored. However, the depth of the tree then only grows logarithmically with the quantity of leaf elements stored which results in a logarithmic search complexity, at least for B-Trees using one single attribute.

4.2.3 Inverted Index

Inverted index structures are directories used to allow the storage of sets of attributes. Actually, the standard relational tree structures may be regarded as specialised cases of the inverted index where the mapping function creates not more than one item key. Hence, the resulting cardinality of the item key set for relational structures is one whereas it may be arbitrarily large for inverted indices.

4.2.3.1 Basic Inverted Index

The inverted index is an indexing approach which can be used to retrieve content elements based on their particular individual components. So, the inverted index can generally be applied to store items in a set representation. An example for these application domains is the storage of texts which are persisted as a set of individual keyword terms. A mapping function $m(t)$ exists which maps from the tuple to the set of indexing keys and an inverse function mapping m^{-1} from one particular index key ik back to the original tuple:

$$\begin{aligned} m(t) &= \{ik_1, ik_2, ik_3, \ldots, ik_m\} \\ m^{-1}(ik) &= \{t_1, t_2, t_3, \ldots, t_i\},\ \forall j \in \{1, \ldots, m\} \end{aligned} \tag{4.44}$$

This is the same mapping as in the relational tree storage approach. However, this case utilises a set-based representation of the outcome of the mapping function whereas the relational storage

points to one representative item key and the inverted index may refer to a set of item keys. The same happens if the set generated by the mapping function for relational values produces sets of cardinality ≤ 1. Hence, the normalised attribute storage may evidently be regarded as a special case of inverted index storage.
The inverted index consists of tuples t which are comprised of the particular tuple attributes a:

$$t = (a_1, a_2, a_3, \ldots, a_m),\ a_i \in N \tag{4.45}$$

The particular elements $m(t)$ of the given tuple t are part of the non-normalised entry set $N = \{nv_1, nv_2, nv_3, \ldots, nv_k\}$.
In an inverted index, an inverse representation of index tuple to representative values exists, which signifies that the tuple (or one identifier) does not point to the particular attributes but the attributes point to the tuple. This also means that the tuple attributes (e.g. a text) are split into individual components and the mapping is applied to them. For this reason, the non-normalised values are separated to a set of distinct parts which then map back to the original tuple. As already described in equation 4.45, each item key derived from one tuple t corresponds to a value $nv \in N$ from the set of all non-normalised entries. Thus, the inverted index constructs posting lists L for each individual non-normalised value nv which can be represented as:

$$inv(nv) = L = \{t_1, t_2, \ldots, t_v\} \tag{4.46}$$

This posting list L points to entries $t \in UT$ that comprise the particular value $nv \in N$ which the list is constructed for. Actually, the posting list only stores the reference to the tuples, either directly to the user table or to some virtual identifier of the tuple itself (pointer). So, it can be rewritten as:

$$L = \{pt_1, pt_2, pt_3, \ldots, pt_v\} \tag{4.47}$$

The list may also be augmented with additional information that can be used, e.g., for ranking, like the frequency or position(s) of the given item nv inside the tuple pointed to by pt_i.
The inverted index itself consists of multiple posting lists. They are set up for each value $nv \in N$. Therefore, the inverted index is composed of all the posting lists which represent a mapping from the values $nv \in N$ to the tuples where they are in:

$$II = \{inv(nv_1), inv(nv_2), inv(nv_3), \ldots, inv(nv_k)\} \tag{4.48}$$

This index then stores a mapping from the non-normalised value to the tuple(s) it is contained in. Each page of the inverted index may only store a portion of the full index.
Each non-normalised value nv may be contained in more than one tuple. Thus, the set of non-normalised values N is constructed by the union of the results of the mapping function:

$$N = \bigcup_{t \in UT} m(t) = \{nv_1, nv_2, nv_3, \ldots nv_k\} \tag{4.49}$$

The way in which the non-normalised value set N grows with the number of values contained in one tuple is uncertain. This analysis can be executed by the use of Heaps' Law [44], an empirical law widely used to describe the relation of items (new ones to the total size of the non-normalised entry set N). It is usually described by:

$$v(q) = K \cdot q^{\beta},\ K, \beta \in \mathbb{R} \tag{4.50}$$

K is a corpus specific constant. The same holds for β. In general, both parameters are determined empirically by studying the respective corpus. v is a function to derive the vocabulary size, which means the size of the set N at a certain point in time. The point in time is described by the value of q with q being the cumulative sum of all entries which were entered to the set of entries at the inspected point of time where also duplicates are included there. Therefore, this law outlines the amount of individually occurring values which are contained with respect to the amount of total values of length q. In many applications, K is between 10 and 100 ($10 \leq K \leq 100$) and β is between 0.4 and 0.6 ($0.4 \leq \beta \leq 0.6$). Based on Heaps' Law it can be derived how the set N grows when a new instance tuple t is added to the inverted index. q in

this case is equal to $\sum_{i=1}^{n} |m(t_i)|$ for a given point in time n. The set size $|N|$ corresponds directly with the vocabulary size $v(q) = |N|$.
Having clarified the number of new index tuples added to the set N of entries at least in a conceptual view, the next question deals with the distribution of tuples in the posting lists, that means the quantity of tuples t which contain one particular element nv. A general rule for the distribution of non-normalised elements to instance tuples can be affiliated when inspecting Zipf's Law [106]:

$$z(rank) = \frac{c}{rank^{\alpha}},\ c, \alpha \in \mathbb{R} \tag{4.51}$$

In this case, the values are ordered because of their occurrence in the corpus. *rank* represents the rank of the particular element. c and α are corpus specific constants which can be deduced empirically. This law describes the amount of occurrences of specific values nv inside the entire set N, which means that it depicts the lengths of the posting lists L inside the index. The entire inspected set is ordered regarding its rank (*rank*) for the representation of this law. Therefore, the number of new values contributed per tuple, due to Heaps' Law (4.50) as well as their distribution to the respective posting lists based on Zipf's Law (4.51) are know.
As seen in equation 4.47, each posting list is set up by using pointers to tuples. With respect to typical database systems, these pointers generally are of equal size (e.g. four bytes each, depending on the system architecture). The length of the list is defined by Zipf's Law (see equation 4.51). On grounds of the power law distribution, it is obvious that there is a small amount of entries which refer to a large amount of tuples and a large amount of entries which refer to a small amount of tuples.
Basically, Heaps' Law states that the quantity of new entries grows more slowly than linearly ($\approx$ with the square root, if $\beta \approx 0.5$) with the number of attribute values of the tuples. This relates directly to the size of the set of non-normalised values N. Zipf's Law, on the other hand, gives relations to the size of the posting lists $L \in II$. So, both of these laws are directly connected with the storage space to be allocated by an inverted index independent from the storage strategy.

4.2.3.2 Tree Based Inverted Index

To make navigation to specific values possible, an auxiliary structure may be set up on top of the non-normalised values nv of the inverted index. In most cases the navigation enabling structure for an index structure of normalised values (as described in section 4.2.2) is, e.g., a B-Tree variant. However, as already explained in section 4.2.1, the length distribution of the set entries is an important attribute if it can be shown that the length of the items is either equally (as for example numbers) or normally distributed (as for example strings). This distribution does not much influence the storage of the particular items. Yet, the distribution of the lists (indicated by Zipf's Law) might affect the storage space depending on the implementation.
According to the storage of the lists, three possible strategies exist:

1. Direct storage of lists together with non-normalised items which means that the postings list is stored in the directory by creating elements comprised of the non-normalised value and the pointer for each $nv \in N$ ("direct storage").

2. Storage of lists separated from the non-normalised items by persisting each non-normalised value inside the directory exactly once and pointing to an external posting list for each non-normalised value which is kept in an extra storage structure ("external storage").

3. Mixed strategy of item 1 and 2 which means that for each non-normalised value a decision based on a pre-defined heuristic needs to be made whether to use the the direct or the external storage strategy ("mixed / hybrid storage"). This indicates that some kind of artificial *limit* value exists which leads to this decision.

To distinguish the best option, it is probably worthwhile to investigate the storage space required by the separate lists and the index in total, respectively. Therefore, the amount of values added by each new tuple t and the distribution of the particular entries nv must be looked at.
If option 2 is chosen, the analysis given in section 4.2.2 applies here. But it must be kept in mind that for each access to the posting list (at least) an additional page / block requires to be loaded per affected value.
In case of the other two options, the analysis does not apply outright as it is possible that additional space is used for the pointer lists directly in the leaf level. In the case of option 1, each entry stores the non-normalised value nv and the list of pointers $L = \{pt_1, pt_2, pt_3, \ldots, pt_v\}$ instantly inside the structure. Therefore, the amount of storage needed is:

$$\begin{aligned} \langle ce \rangle &= \langle nv_j \rangle + \sum_{i=1}^{v} \langle pt \rangle \\ \Rightarrow \quad \langle ce \rangle &= \langle nv_j \rangle + v \cdot \langle pt \rangle \end{aligned} \tag{4.52}$$

For option 3, the storage of one entry can be differentiated by the artificial *limit* value as:

$$\langle ce \rangle = \begin{cases} \langle nv_j \rangle & \text{if } v \geq limit \\ \langle nv_j \rangle + v \cdot \langle pt \rangle & \text{if } v < limit \end{cases} \tag{4.53}$$

The total set size of the items to be stored in the inverted index is thus comprised by the total quantity of items, which can be derived from Heaps' Law, as it builds an estimation for the total amount of individual values, and Zipf's Law which allows an estimation about the amount of references pointed to by one individual item of the entry set N.

4.2.3.2.1 External Storage of Pointers External storage of pointers probably is the most trivial case for an inverted index. If the references are stored in additional storage spaces, which means the posting lists L are stored in a separate place, the amount of values to be stored inside the leaf level of a tree is equal to $|N|$. Therefore, the count of leaf node pages P_l required to store the particular references can be expressed as:

$$|P_l| = \frac{|LE|}{n_l} \tag{4.54}$$

The above equation may be assumed if a certain distribution (where a representative location parameter can be derived from) can be applied to the value sizes in N. The distribution to inner nodes of, e.g. a B-Tree, can be done analogously to the one given in 4.2.2.2. The length of the pointer lists is distributed as described in Zipf's Law. This leads to the situation that values $nv \in N$ exist which have very large lists and others which have very short pointer lists. However, in each case, at least one additional page containing a portion of the pointer list must be loaded for each operation (e.g. searching / insertion / ...).
This structure is optimal for a high amount of occurrences of a certain term because in this case, it requires exactly one entry in the directory per value $nv \in N$. The remaining references are persisted in external posting lists. Hence, the directory tends to stay dense. This also directly relates to the retrieval performance. In this case, for single term queries, the asymptotic retrieval complexity is

$$\mathcal{O}\left((\log |LE| \cdot s) + m\right) \tag{4.55}$$

with s being the number of terms and m the number of occurrences. Actually, this formula applies to all retrieval models. However, it depends on the size of the leaf entry set. The size of the leaf entry set in this case is $|LE| = |N|$ because for each non-normalised value, there is only one entry inside the directory. With respect to the retrieval of the B-Tree, this complexity is optimal. This is also true for large amount of occurrences of one non-normalised value. Unfortunately, this is not true for a small frequency. In this case, for each value, (at least) an additional page must be accessed.

4.2.3.2.2 Direct Storage of Pointers When storing the pointers directly in the leaf nodes, the pointers themselves might use a large amount of space which lowers the effective branching factor (at least of the leaf nodes) substantially. There is a function $rank(nv_i) = rank_i$ which derives the rank from each individual unique value nv for Zipf's Law values. So, the amount of pointers which are contributed by an individual value can be rewritten as (see 4.51):

$$z_i = z(rank(nv_i)) = \frac{c}{rank_i^{\alpha}} \tag{4.56}$$

Then z_i is the Zipf's distribution value for item nv_i with the rank $rank_i$ extracted from $rank(nv_i)$. If one content element intended to be stored in the directory is comprised by the actual non-normalised value in conjunction with the pointer(s), the total amount of storage for one particular value can be written as:

$$\langle ce \rangle = \langle nv \rangle + z \cdot \langle pt \rangle \tag{4.57}$$

In some cases, however, the amount of pointers to be stored exceeds the available space in one page. Then, again, multiple pages must be loaded for one particular item. As the pointers are also stored directly inside one page, the depth of the tree increases, too.
Values only pointing to a small amount of documents do not contribute much to the total size of leaf entry set LE whereas values having a low rank contribute much to the size of the required leaf entries storage space. Therefore, values occurring seldom do not affect the branching factor much whereas values appearing frequently lower the branching factor extremely.
The retrieval complexity (see equation 4.55) is solely dependent on the size of the leaf entry set. It is composed of the actual non-normalised values in direct combination with the pointers in this application. Hence, the size can be expressed as:

$$|LE| = \sum_{i=1}^{|N|} z\left(rank\left(nv_i\right)\right), \; nv_i \in N \tag{4.58}$$

That means that the leaf entry set contributes more than one element per non-normalised value due to this storage strategy. The size of the leaf entry set is constructed by the amount of occurrences per non-normalised value in this case. As the directory stores more values, a tree tends to become higher (regarding the number of levels). Unfortunately enough, the effective branching factor decreases because the effective information about the set N stays the same but the leaf entry set is at least as big as the cardinality of set N ($|LE| \geq |N|$). As this is based on Zipf's Law, there is a small quantity of terms which contribute a lot of occurrences and a large one which adds a small number of occurrences. Here, the size of the leaf entry set is larger than the set of non-normalised values. For the large number of seldom occurring values, this method seems to be superior to the external storage because no extra pages need to be loaded during the retrieval. Regrettably, this is not true for the small number of frequent values which are also stored directly inside the directory.

4.2.3.2.3 Mixed Approach of Storage of Pointers The third approach for the storage of the posting lists is a mixed approach. This approach introduces an artificial upper bound *limit* which can arbitrarily be defined (e.g. the size of a block / page). It moves the posting lists of values to an external storage space, if the frequency is greater than this limit ($z > limit$), and, for the rest stores the pointers directly inside the directory. This seems to be the most reasonable approach, if *limit* is set up probably. In this case, values occurring frequently and not contributing much to the total storage space required by one item are stored directly inside the directory, whereas items occurring more often than *limit* times are stored in an external storage space to keep the directory small. This option combines the benefits of the direct storage with the external storage. The external storage suffers from situations where there are only small frequencies of items. In these cases, an additional page must always be loaded, during insertion as well as retrieval, even if there is only one single reference. Then, a more direct access is desirable by storing the references directly at their place inside the directory. If the amount of

references is high instead, the external storage is more favourable because values which are present in a high frequency bloat up the directory. An example for setting the *limit* parameter could be the available page storage $\langle p \rangle$. Whenever the sum of the storage requirements of the particular content elements $\langle ce \rangle$ is greater than the available page block size $\langle p \rangle$, an additional page must be loaded. Hence, these items occur frequently and additional pages must be modified / retrieved during the particular operations. That means that this option combines the benefits of a fast retrieval in case of low frequently occurring non-normalised values by the direct loading approach and the directory kept small when high frequently occurring values are externalised. This indicates that by means of this alternative the worst cases for both options can be avoided if the *limit* parameter is configured appropriately.

4.2.4 Hybrid Index

The hybrid index is a structure to index tuples of both, normalised and non-normalised values. Therefore, it uses point values in the normalised part and textual values in the non-normalised parts. It can, e.g., be used for spatial keyword queries which it supports efficiently. Besides spatial keyword support for setup in geographic information retrieval systems, also combinations of standard relational data as used in enterprise content management systems (dates, prices, ...) connected with non-normalised data like texts, image features or similar data are imaginable.
So, formally, the hybrid index gets to store tuples t comprised by relational / normalised and non-relational / non-normalised components:

$$t = (rp, np) \tag{4.59}$$

The relational part (rp) represents in this case one (or more) normalised value(s) $rv \in R$, which means that it is simply a point value. The non-normalised part (np) stands for one non-normalised value. It is a text composed of a set of words $nv \in N$. A function $w(t)$ exists that is responsible for generating the extraction of the non-normalised values from the tuple content. Furthermore, there is a function $p(t)$ for retrieving the normalised content, i.e. the mapping function which is in charge of the extraction of the respective tuple contents. Thus, it can be stated that:

$$\begin{aligned} w(t) &= \{nv | nv \in N \wedge nv \in t\} \\ w^{-1}(nv) &= \{t | nv \in w(t)\} \\ p(t) &= \{rv | rv \in R \wedge rv \in t\} \\ p^{-1}(rv) &= \{t | rv \in p(t)\} \end{aligned} \tag{4.60}$$

The mapping function $w(t)$ only treats the non-normalised part whereas $p(t)$ cares about the normalised part of the tuple to be stored. Inverse mapping functions exist, as well. The hybrid index to be inspected here stores tuples with a normalised and a non-normalised value part. The index construction is set up as demonstrated in figure 4.3. Currently, an initial inverted index exists which is the initial storage structure to save the index tuples. It stores the values $nv \in N$ present in np of object t. One important property is the existence of an artificial upper bound *HLimit* which separates the set N into high (N_h) and low (N_l) frequent value subsets. Elements in N_l occur less frequently than *HLimit*.

$$\begin{aligned} N_h &= \{nv | nv \in N, |w^{-1}(nv)| > HLimit\} \subseteq N \\ N_l &= \{nv | nv \in N, |w^{-1}(nv)| \leq HLimit\} \subseteq N \end{aligned} \tag{4.61}$$

HLimit as an input parameter is an artificial upper bound which may be set arbitrarily but is, generally, based on the corpus used as an input set. The separation of items is described in further detail later on. The initial inverted index only stores low frequently occurring values directly by referencing them to the document heap, which means that the pointers go outright into the document heap from there. High frequently occurring values are stored in the main component of the index, the hybrid R-Tree. This R-Tree contains the information stored in the relational part (rp) of the objects. It stores, besides the normalised values, a bitlist which

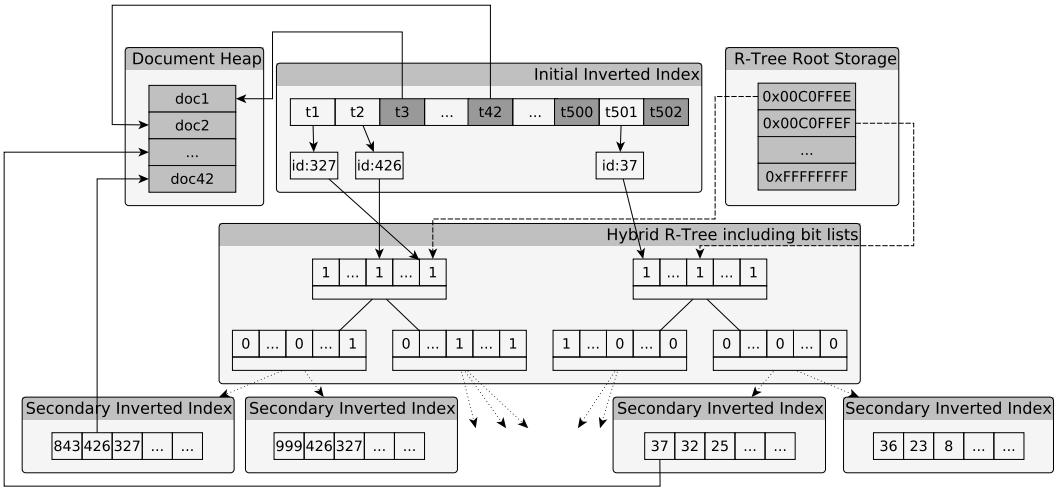


Figure 4.3: Conceptual Overview of the Hybrid Index Structure (adopted from [41, 62, 63])

indicates the presence or absence of a non-normalised value, too. When the amount of references of a given non-normalised item is greater than *HLimit*, which means that $z(rank(n_i)) = z_i > HLimit$, the relational part *rp* of all object references which contain the particular non-normalised value nv_i are moved to the hybrid R-Tree and the values are directly stored there.
Each of the non-normalised values $nv \in N$ which occur more than *HLimit* times get assigned a unique id value which is then used inside the bitlist to identify these attributes. So, the initial inverted index also separates the total set of non-normalised values into two disjoint subsets $N_h \subseteq N$ and $N_l \subseteq N$ with $N_h \cap N_l = \emptyset$. Set N_h contains all high frequently occurring values which have a frequency value greater than *HLimit*. The set N_l contains all low frequently occurring values which have an occurrence frequency $\leq HLimit$.
The posting lists *L* of each value point to a structure called "document heap". This document heap is responsible for managing the indirect storage of the pointers back to the original tuple. The content elements of this document heap are the relational part *rp* of the respective objects pointed to by the pointers stored, there. This relational part is inserted to the hybrid part of the index if there is an item in the set of non-normalised attributes of the respective object which occurs more than *HLimit* times. Thus, the normalised values are also separated into high and low frequently occurring ones based on the non-normalised values they coocurr with inside a tuple. This constructs the following set:

$$R_h = \left\{ rv | rv \in R, \left(\exists nv \in w\left(p^{-1}\left(rv\right)\right) \wedge nv \in N_h \right) \right\} \subseteq R \tag{4.62}$$

The hybrid index part is composed of the normalised value (a point value in this case) and a bitlist. This bitlist represents the sets of entries $nv \in N_h$ valid for the subtree pointed to by the associated pointer of the page elements. The bit number is derived sequentially, if a new tuple is added which makes one particular non-normalised value $nv \in N$ move from N_l to N_h. The non-normalised value *nv* then gets assigned the particular next free bit index number and can be checked for presence inside the hybrid index by verifying whether or not the respective bit is set in the element content.
The final destination for the references to the document heap is the secondary index structure which stores the mapping from bit numbers to the entries of the document heap or the original tuple in the user table, respectively. This secondary index is the ultimate destination of the items because the values $nv \in N_h$ are stored there, finally, after having traversed the tree. One value is stored at a particular secondary index, if one tuple *t* has a relational component *rp* equal to the entry which points to the secondary index. Hence, this relation can be expressed as:

$$nv_h(rv) = \{ nv | nv \in N_h, \exists t \in UT : rv \in p(t) \wedge nv \in w(t) \}, \ \forall rv \in R_h \tag{4.63}$$

In this way the set of non-normalised values valid for a given normalised value is formed. This set is constructed per normalised value and thus varies in contents as well as in size.

4.2.4.1 Initial Inverted Index

The initial index stores information about all non-normalised values. However, there is the upper bound *HLimit* which decides whether the frequency of a certain value is low or high. Hence, a certain value *nv* may only appear in one of the subsets N_h or N_l. Consequently, the necessary information stored in this area is either one non-normalised value and its posting list *L* directly, if the item occurs less than *HLimit* times or the non-normalised value and an identifier which makes it re-findable inside the hybrid index tree. So, the storage of one content element here, can be summarised as (*bi* is one bit index represented as integer value, e.g. 4 bytes):

$$\langle ce \rangle = \begin{cases} \langle n \rangle + z \cdot \langle pt \rangle & \text{if } n \in N_l \\ \langle n \rangle + \langle bi \rangle & \text{if } n \in N_h \end{cases} \tag{4.64}$$

The entries stored inside the initial inverted index are non-normalised values which have the same properties and can thus be treated exactly as described in section 4.2.3. Whether or not to include the mixed storage approach depends on the set up of *HLimit* as this limit also represents a pre-defined cut-off point in Zipf's Law [106] and therefore separates values with a low frequency from those with a high frequency.
The size of the set *N* is also the same as described in section 4.2.3:

$$|N| = v(q) = K \cdot q^{\beta} \tag{4.65}$$

However, the size of posting lists to be managed by this component differs. This component only copes with the posting lists of the values of N_l thus resulting only in short posting lists depending on *HLimit*.
For this reason, a mapping function $\psi(nv)$ exists which decides whether to put the respective item *nv* into the hybrid part of the index or if it already resides inside the hybrid part of the index. By the use of this mapping function, the elements of set N_h are arranged in a specific order. based on the result of function ψ which describes the bit index number, each of the elements in N_h one value is allocated to. It is assigned sequentially based on the event of time when the posting list associated with one non-normalised value is longer than *HLimit*. It is defined recursively as:

$$bi = \psi(n_j) = \begin{cases} \psi(n_{j-1}) + 1 & \text{if } |L| > HLimit \Rightarrow n_j \in N_h \\ 0 & \text{if } |L| \leq HLimit \Rightarrow n_j \in N_l \end{cases} \tag{4.66}$$

Only if $\psi(n)$ is $\neq 0$, the item is processed further inside the hybrid index logic, otherwise it is handled directly at this place. The resulting value of ψ is also persisted, besides the actual non-normalised value inside the initial inverted index. Thus, if a value resulting from a previous call to the mapping function is already present inside the initial inverted index in conjunction with the non-normalised value, it is obvious that the processing of the respective value (either during reorganisation or retrieval) must proceed inside the hybrid R-Tree.
Regarding the set sizes to be managed inside the initial inverted index it can be stated that both sets, N_l and N_h must be taken care of. However, only one value per element in N_h needs to be managed as further references are stored somewhere inside the hybrid R-Tree. Because of N_l saving only the values occurring low frequently inside the entire index, there should not be any issue regarding retrieval performance. Therefore, the set size of the leaf entries to be handled by the directory of the initial inverted index can be depicted from:

$$|LE| = |N_h| + \sum_{i=1}^{|N_l|} z\left(rank\left(nv_i\right)\right), \; nv_i \in N_l \tag{4.67}$$

In the above equation, the respective outcome of the Zipf function tends to be a very low number because the values comprising high frequencies are members of the set N_h. Hence, the N_l part of the equation potentially contributes a lot of items, but only with a very low frequency.

4.2.4.2 Hybrid R-Tree

The hybrid R-Tree manages objects of both types, which means that it cares about the relational part *rp* and the non-relational part *np* of the tuple simultaneously. In the hybrid index part, each entry is composed of the normalised value of a tuple *t*, if this tuple contains (at least) one non-normalised value which occurs more than *HLimit* times and a set content. In this case, the set content is represented as a bitlist. Inside this bitlist, the value 0 represents the absence of a certain item whereas the value 1 represents its presence. The bit for the respective item is derived from the bit index *bi* assigned to the particular item the first time the amount of references to that particular item overflows *HLimit*. It can be retrieved via the mapping function $\psi(nv) = bi$.
The base index used here is an R-Tree which stores points in the leaf level and ranges, composed if a lower and an upper point, in the inner levels. Therefore, there is a differentiation between content elements of inner and leaf nodes. The size of the inner (ic_i) or leaf content elements (ic_l) is calculated via (*bl* represents the length of the bitlist):

$$\begin{aligned} \langle ic_i \rangle &= \langle range \rangle + bl \cdot \langle bit \rangle \\ \langle ic_l \rangle &= \langle point \rangle + bl \cdot \langle bit \rangle \end{aligned} \tag{4.68}$$

It must be noted that the required storage space of the storage element $\langle bit \rangle$ in general implementations is 1 bit. As already explained, the size of a *range* with respect to the storage space required is nearly twice as big as the size of a *point*. Hence, for further size estimation, it may be assumed that it is more safe to calculate with ranges.

$$\begin{aligned} & \langle p \rangle && = \langle h \rangle + n \cdot (\langle o \rangle + \langle pt \rangle + (\langle range \rangle + bl \cdot \langle bit \rangle)) \\ \Rightarrow \quad & n && = \frac{\langle p \rangle - \langle h \rangle}{\langle o \rangle + \langle pt \rangle + (\langle range \rangle + bl \cdot \langle bit \rangle)} \\ \Rightarrow \quad & bl && = \frac{\langle p \rangle - \langle h \rangle - n \cdot (\langle o \rangle - \langle pt \rangle - \langle range \rangle)}{n \cdot \langle bit \rangle} \end{aligned} \tag{4.69}$$

The three equations in equation 4.69 show the interdependence of the three values of $\langle p \rangle$, *n* and *bl*. So, it is possible for any given two of them to express the third variable. Generally, the page size $\langle p \rangle$ is set by the database. That is why either the length of the bitlist or the amount of items in one page may be set arbitrarily (under certain pre-conditions so that they stay feasible). An optimal R-Tree has a best case search complexity of $\mathcal{O}\left(|LE|^{(d-1)/d}\right)$ with *d* being the number of dimensions. The growth of the R-Tree regarding the amount of inserted elements is linear whereas the height, in general, grows logarithmically.
Based on the fact that the hybrid R-Tree works on normalised values which appear in tuples containing high frequently used non-normalised values, the growth of the set R_h of normalised values to be handled by the R-Tree is similar to the entire set *R*. That is why high frequently used non-normalised values tend to occur in a large set (or even all) tuples stored inside the user table. Therefore, the size of the set R_h at a given point in time can be described as:

$$|R_h| \approx |R| \tag{4.70}$$

Normally, the growth of the set can also be described with Heaps' Law [44] ($v(q) = K \cdot q^{\beta}$, $K, \beta \in \mathbb{R}$), however, the parameters tend to be different in comparison to the non-normalised value part.

4.2.4.3 Secondary Index

The secondary index is the final destination for the entries. It is referred to from each normalised value inside the leaf level of the hybrid R-Tree. In this index part, the non-normalised values are stored if they belong to the set N_h. They are distributed to the particular normalised value entries in the hybrid part if the respective non-normalised value is in the set returned by the function $nv_h(rv)$ (see equation 4.63) which forms the intersection between both parts. In this

structure, the storage of all the values may be omitted as a unique number (*bi*) is assigned to each $nv \in N_h$. That means that this structure, in combination with the normalised value it is attached to, forms the intersection between the normalised and non-normalised value parts of the hybrid index. Yet, it only returns the non-normalised content mappings back to the tuples. In case of a query it is already obvious that the respective normalised value one secondary inverted index is attached to is a valid candidate for fulfilling the normalised query part and also contains the terms identified via their bit index numbers satisfying the non-normalised search condition. The size of the set to be managed by one respective secondary inverted index is already given in equation 4.63. Unfortunately, because the particular non-normalised values are distributed through the R-Tree and only overflow (regarding the quantity of references) *HLimit* in the initial inverted index part, similar properties apply to the subset $nv_h(rv)$ as seen in the general inverted index. Hence, it is probably meaningful to apply an appropriate strategy (such as used by the general inverted index).
The entries stored inside the secondary index consist of the unique number (*bi*) in combination with a pointer back to the original user table tuple. Thus, the space required for one content element is:

$$\langle ce \rangle = \langle bi \rangle + \langle pt \rangle \tag{4.71}$$

This is substantially smaller than in the remaining storage spaces and leads to a potentially high branching factor depending on the strategy chosen for the storage of the elements.

4.2.4.4 Setup of *HLimit*

As already discussed, the sets N_h, N_l, R_h as well as the set of non-normalised values valid for a given normalised value ($nv_h(rv)$) depends on the setup of *HLimit*. This artificial user defined boundary is installed to separate the set N into two parts. One of the subsets consists of the non-normalised values referenced more than *HLimit* times and the other one of the complementary set. This limit is, on the one hand, introduced to margin the amount of non-normalised values inside the hybrid index to keep the size of the bitlist manageable and, on the other, because it may simply be more efficient to scan the resulting elements linearly than to enter the search procedure to the hybrid R-Tree. Hence, this limitation is applied after the manipulation or search queries at the initial inverted index. Thus, only frequently occurring non-normalised values will be treated in the hybrid R-Tree.
This leads to the discussion about the question when to regard a value as frequent. It is therefore probably worthwhile investigating the asymptotic complexity of search queries relative to the corresponding handled storage structures. The associated search time complexities are already noted inside the particular paragraphs describing the respective storage structures.
In this inspection, the persistence unit named "document heap" plays an important role. Inside the document heap, all normalised values are stored. This comes from the fact that non-normalised values solely occurring in the set N_l need to be checked for the normalised values sequentially. Hence, a designated storage structure must exist caring about these elements. This is the task of the document heap which contains the raw mapping of tuples to the normalised values they contain. Initially, before the amount of references to one non-normalised value $\left|w^{-1}(nv)\right|$ is greater than *HLimit*, a pointer to the document heap is stored inside the initial inverted index which makes the retrieval of these values possible. Hence, when it comes to retrieval, all normalised values stored for one document there must be checked. For one document this makes a total effort of:

$$\sum_{i=1}^{k} |p(t_i)| \tag{4.72}$$

This is due to the fact that each individual normalised value from each tuple (t_i) needs to be checked once for presence inside a search condition. The number of documents in this case is k.

The worst case for the retrieval of elements from the document heap is that each tuple points to the set of normalised values with maximum cardinality. Then the retrieval effort at this stage is:

$$k \cdot \max_{t \in UT} (|p(t)|) \tag{4.73}$$

This is only true, if no possibilities for skipping already performed comparisons exist.
For multiple terms, also multiple sets of documents must be inspected:

$$\sum_{j=1}^{s} \sum_{i=1}^{k} |p(t_{ji})| \tag{4.74}$$

Yet, the omission of checks by the use of more appropriate document intersection techniques may lead to a reduction to the minimum number of occurrences of one document in the non-normalised part. Hence, equation 4.73 is also true for the worst case inside the document heap.
This search effort must be compared to the one performed in the hybrid index. The initial inverted index comprises a search effort of:

$$\left(\log \left(|N_h| + \sum_{i=1}^{|N_l|} z(rank(nv_i)) \right) \cdot s \right), \; nv_i \in N_l \tag{4.75}$$

However, this effort may be neglected for the following analysis because it is present in both query types (document heap and hybrid R-Tree) and does consequently not play any role for the retrieval of elements.
The relational structure might be a B-Tree comprising a logarithmic time complexity or an R-Tree which cannot provide this in the general case. The R-Tree may achieve such a time complexity if it is appropriately modified and pre-calculated (see [37]). Hence, it may be assumed that there is a relational storage structure whose worst case search time complexity is:

$$\log |R| \cdot q \tag{4.76}$$

In the above equation, q is the size of the result set, which means the number of normalised values included in the search region. Besides, R is taken as approximation of the set size because in the worst case where the projection function $p(t)$ of each tuple returns the entire normalised value set R, $R_h = R$.
As an addend to this time complexity, the secondary inverted index must be inspected. This access structure is loaded for each normalised value which is inside the respectively affected query object. Only valid secondary indices are accessed if the normalised condition is fulfilled and the non-normalised condition has a chance to be fulfilled. The bitlist indicating the presence or absence of a non-normalised value is checked for the existence of the respective bit numbers during traversal. Once again, this index structures is an inverted index which stores assignments from bit index numbers representing non-normalised values to pointers to documents. Hence, the retrieval effort is:

$$(\log |nv_h(rv)| \cdot s) + m \tag{4.77}$$

In the above equation, s is the quantity of query keywords and m is the size of the final result set. Therefore, the summarised retrieval effort of the hybrid R-Tree can be expressed as:

$$\log |R| \cdot \sum_{i=1}^{q} ((\log |nv_h(rv_i)| \cdot s) + m_i) \tag{4.78}$$

To constitute the bounds and a method how to set the limit correctly, the search effort for the document heap must be compared with the total labour spent inside the hybrid R-Tree. This results from the fact that either the document heap is scanned linearly or the search continues inside the hybrid R-Tree. For this reason, the basic question is to find out when it is more efficient to search in the hybrid index than to determine the final result set from the document heap.

Generally, both retrieval efforts are targeted towards the set R of normalised values. The hybrid R-Tree has a logarithmic complexity regarding this set (at least in an optimally packed environment). The document heap fulfils queries using k scans over the maximum number of normalised values per tuple. k is limited by the number of references to one term. Thus, k is actually a factor of $\left|w^{-1}(nv)\right|$ for one non-normalised value. The relation to be inspected is:

$$\begin{aligned} & \log|R| \cdot \sum_{i=1}^{q}\left(\left(\log\left|nv_h\left(rv_i\right)\right| \cdot s\right) + m_i\right) \leq k \cdot \max_{t \in UT}\left(\left|p\left(t\right)\right|\right) \\ \Rightarrow \quad & \log|R| \cdot \sum_{i=1}^{q}\left(\left(\log\left|nv_h\left(rv_i\right)\right| \cdot s\right) + m_i\right) \leq \max_{nv \in N}\left(\left|w^{-1}(nv)\right|\right) \cdot \max_{t \in UT}\left(\left|p(t)\right|\right) \end{aligned} \tag{4.79}$$

The inequation which must be decided when considering *HLimit* is shown in equation 4.79. This limit is actually equal to the factor $\max_{nv \in N}\left(\left|w^{-1}(nv)\right|\right)$ given in the inequation.
Therefore, in total to isolate *HLimit* in the above equation the following must be applied:

$$\frac{\log|R| \cdot \sum_{i=1}^{q}\left(\left(\log\left|nv_h\left(rv_i\right)\right| \cdot s\right) + m_i\right)}{\max_{t \in UT}\left(\left|p(t)\right|\right)} \leq \max_{nv \in N}\left(\left|w^{-1}(nv)\right|\right) \tag{4.80}$$

For the exact calculation in this case, the set sizes are not sufficient. Basically, a rough estimation can only be given here. The storage sizes must additionally be taken into account. Regarding the R-Tree and the B-Tree used to access the inverted index, the branching factor may be calculated by dividing the available page size through the size of one individual storage item. Hence, for the R-Tree, the number of elements to be found may be expressed as:

$$\log_{bf_r}|R| \tag{4.81}$$

The branching factor bf_r in this instance may be calculated by inspecting the page size ($\langle p \rangle$) and the item content size of the R-Tree. That means that the branching factor of the R-Tree is $bf = \frac{\langle p \rangle}{\langle ic \rangle}$. The same calculation may be applied for the B-Tree.

$$\log_{bf_b}|N| \tag{4.82}$$

As to the retrieval of the document heap, also the item content size must be inspected. Therefore, it is necessary to multiply the set size with the item content size as well $\max_{t \in UT}\left(\left|p(t)\right|\right) \cdot \langle ic \rangle$. Altogether, the retrieval effort is:

$$\frac{\max_{t \in UT}\left(\left|p(t)\right|\right) \cdot \langle ic \rangle}{\langle p \rangle} \tag{4.83}$$

This leads to the following final result:

$$\frac{\log_{bf_r}|R| \cdot \sum_{i=1}^{q}\left(\left(\log_{bf_b}\left|nv_h(rv_i)\right| \cdot s\right) + m_i\right)}{\max_{t \in UT}\left(\left|p(t)\right|\right)} \cdot \frac{\langle p \rangle}{\langle ic \rangle} \leq \max_{nv \in N}\left(\left|w^{-1}(nv)\right|\right) \tag{4.84}$$

The final outcome of equation 4.84 describes the dependencies of the respective set sizes. Still, a minimisation must be done in order to retrieve the final result. This is required because the actual limit which is found on the right side of the equation is also contained in the size of the non-normalised value set valid for one normalised value ($nv_h(rv)$). The size of this set depends on the size of *HLimit* and thus, contained on both sides. In order to finally determine *HLimit*, it is thus inevitable to minimise the equation.

Note that some excerpts, especially, the main results about the proper setup of *HLimit* are present, in a similar fashion, in [62].
Summarising, this chapter answers the research question regarding a generalised model approach by defining the only feasible option for model based building a hybrid access structure. The basic research question is given in subsection 1.2.1.3.

5 EFFICIENT REORGANISATION OF HYBRID INDEX STRUCTURES SUPPORTING MULTIMEDIA SEARCH CRITERIA

As already explained in section 1.2, the main focus of this work is to optimise hybrid index structure reorganisation algorithms. Therefore, techniques to analyse the program as well as specific parts must exist. General profiling may be applied here, as well as further techniques to analyse the runtime of the program in total. Static code analysis must be carried out additionally in order to validate the profiling results for the chance of optimising a certain algorithm. Besides, also general attributes of databases and index structures must be kept in mind to find ways of optimising the algorithms properly.
That is why it is necessary to build a test environment which can run predefined test cases and measures their performance.
This chapter describes the main research work including the setup of the test suite which must be reasoned out at the beginning as well as the particular optimisation approaches.
Rigorous methods from the knowledge base are utilised (see 1.2.2.1.5) in the measurement and construction part as well as the evaluation and design. It must be noted that the regulative cycle described in subsection 1.2.2.2 just as the build and evaluate loop described in section 1.2.2 comes into effect. In general, either creative solutions in the style of a trial-and-error search are evaluated or already present knowledge is applied to the optimisation procedure. Summarising, this chapter delineates the search process (see 1.2.2.1.6) needed to find an improved variant of the reorganisation algorithms.

5.1 SET UP OF THE TEST SUITE

For measuring the performance in a reliable and reproducible way, a well defined test environment must exist by means of which the details about the internal states of a program can be monitored. Referring to index structures, some specific parameters must be selected which represent the internal state. The time to load a certain entry from a hard disk is one of the most important properties when considering database development. Nowadays, this time frame is still in the range of milliseconds, whereas access to main memory is done in nanoseconds.

Therefore, a crucial task for database optimisation is the effort to limit the accesses to hard disks.
However, this is not the only parameter of interest when constructing a test suite to monitor the performance of a database index structure. Obviously, the time needed for certain function calls or their sequences, is a very important feature as there are inefficiencies in the parts implemented in main memory, too.
For being able to monitor all the important features inside the database, a test suite was constructed which was continuously updated if new features were to be measured in order to produce reliable and reproducible results. Thus, the detailed construction of such a test suite as well as defined input data used to measure the performance had to be created. Besides the features which can be measured, analysis techniques are also required which, after a certain measurement, have the possibility to interpret the measured features and give hints about potential candidates to be optimised in a particular optimisation step.
This section describes the construction of the test suite as well as all input data and measurement techniques used to provide hints to optimise certain parts of the program.

5.1.1 Test Suite

The test suite is a really significant part of this thesis. This component measures all the data to be collected during the program run. Therefore, it is also essential to define the details precisely. It is not only important to detail what will be measured but also how the measurement will take place. This implies that the sequence of actions must also be precisely defined in advance to generate reproducible results.
The runtime of the particular methods is the most important feature to extract here, because the main target is to improve the insertion performance of the hybrid index structure. However, not only the runtimes are important as they strongly relate to other features of an access structure. As already discussed, for a database which is mainly a hard disk centric application, it is also necessary to check for page accesses. Thus, the test suite needs to measure other features like disk page accesses, too. That means that the test suite must be able to gather a wide variety of properties during insertion procedures. Nevertheless, as the main goal of this work is to enhance the insertion performance while keeping the search performance it is not only important to measure the insertion performance but also the queries.
The test suite is created in order to extract all these key features. It would also be interesting to augment the pre-compiled source code with further instructions without having to modify the source code for the target of the measurements.
After evaluating techniques like employing a profiler, e.g. using the JVM TI[1], which has only the capability of monitoring function calls, among others, the decision was taken to implement the functionality of the test suite applying aspect oriented programming (see, e.g. [60]). Based on the fact that the already existing approaches, see e.g. section 2.1.4, lacked at least some of the functionalities desired for the analysis of the program paths, a new test environment for dynamic program analysis and evaluation was created. The aspect oriented programming paradigm does not rely on object oriented class hierarchies but counts on functionalities, too. Each of the so-called aspects of a program fulfils a certain functionality. They are subsequently combined to a program by executing them whenever they are required. This results in a loose coupling of aspect components. Thus, additional functionality may be just hooked into a program when it is needed. This technique makes it easy to augment the original source code with extra capabilities. One commonly used example is the aspect of logging for a program. There is often the need for tracing inside programs which means that for each function call a message should be written into a file or the console. One option to achieve this is to place the calls to a logger by hand into the source code at each function call. However, two basic problems arise. On the one hand, putting the logging information into the source code by hand bloats the code as each method contains, at least, two additional function calls where just log messages are written in, manually. On the other, there is the probably high chance that developers forget to insert the log

[1] `http://docs.oracle.com/javase/6/docs/technotes/guides/jvmti/` accessed 2013-28-06

messages, by hand. Additionally, the loggers must be configured properly and write the information at a pre-defined logging level, e.g. TRACE, INFO, DEBUG. If this information is added manually, there is again a high potential for making mistakes in there.
Aspect oriented programming overcomes these weaknesses by providing the possibility to hook these kinds of functionalities into the existing program. As the database and the remaining code are implemented in Java, the AspectJ[2] library is employed here. Specific points where to augment the functionality may be defined as `JoinPoints` where `PointCut` definitions specifying circumstances when to execute the advice may be applied, which is the real functionality to be executed.

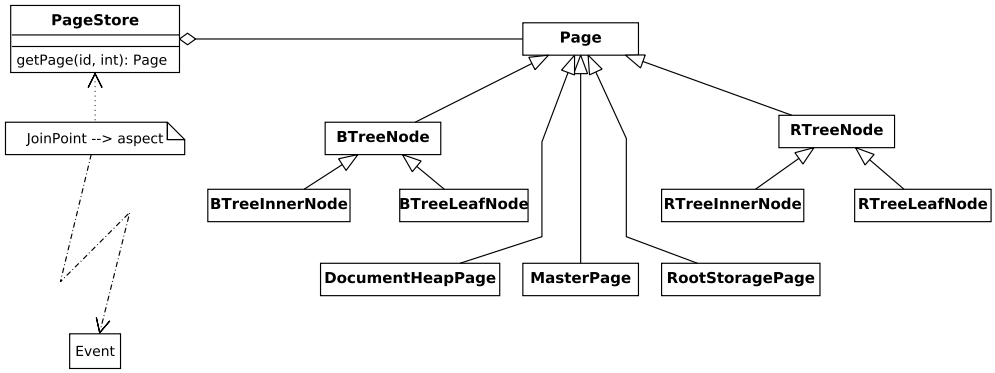


Figure 5.1: Representation of a JoinPoint intercepting Calls

An example of a "JoinPoint" can be seen in figure 5.1. It already shows a conceptual overview as used inside the real test suite. The `PageStore`, generally, manages accesses to all page types inside the database. Therefore, in order to realise when a page is loaded, a `PointCut` is set up using the `JoinPoint` at the `getPage` method which results in the advice that the page access should be noticed. This conceptual overview also shows the particular page types to be loaded inside the database and the advice attached to it registers an access to the particular page depending on its type.
The aspect to notice the page types, however, is registered to one particular function. Instead of registering to specific functions, however, this technique may also be used together with wildcards to listen to a set of method prototypes or even all. In this work, the entire program is the subject of investigation and thus, wildcard `JoinPoint` definitions are used to register calls of all functions inside the execution of methods of the hybrid index.

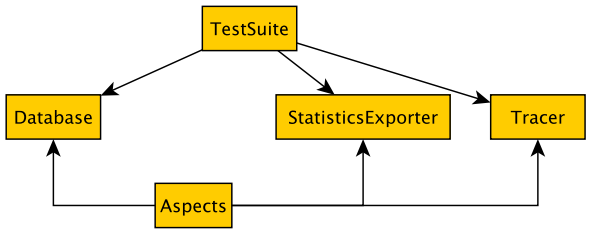


Figure 5.2: Overall Component Setup of the Test Suite including Aspect Interface

An overall overview about the test suite can be seen in figure 5.2. Basically, the `TestSuite` controls the testing procedure. It starts and finalises the components and executes the particular test cases in a predefined order. The `StatisticsExporter` component mainly cares about statistics generated, e.g., from loading pages or a few other test parameters explained in subsection 5.1.2. The `Tracer` component cares about logging all function calls using wildcard `JoinPoint` definitions in a pre-defined way. The `Database` is the main part of the test suite as

[2] `http://eclipse.org/aspectj/`, accessed 2013-28-06

it is actually the component under test here. The test suite therefore must ensure that the database is correctly started and everything is properly set up in the environment.
In the first test runs it was noted that tracing the function calls continuously on the one hand takes much longer time, as the amount of function calls doubles because of having to execute the tracing code as well and on the other produces very large trace files described later on. Therefore, it is possible to start the test suite with different configurations. The decision was to insert a couple of documents into the database without tracing functionality and then one single document with tracing continuing again without. This leads to regularly occurring measurement points during the entire run of the test suite. Consequently, a mechanism was built allowing the test suite continuing its work after a restart.
The measurements are run in a way that always 25 documents from the collection, described in subsections 5.1.3.2 and 5.1.3.3, are inserted. After 25 documents queries are carried out to monitor the behaviour of searching, too and another 25 documents are inserted afterwards followed by queries. Having inserted 50 documents, the test suite is stopped and one document is inserted while enabling the tracing functionality. Therefore, each 51st document inside the collection defines a measurement point for function call tracing. However, the parameters of measurement may also be arbitrarily changed. There are a couple of additional options for the test suite environment which may be changed for testing purposes. Yet, they are not very important and thus left out in this description.
Concluding it can be said that the test suite generally starts and stops all components related to the test cases to be executed. Therefore, this is a very central component for the tests to be carried out and ensures a smooth execution of the test cases by starting and stopping all related components reliably.

5.1.2 Variables and Parameters

There is a list of variables and parameters measured by the test suite during the execution of the tests. Table 5.1 lists all aspects and the type of information they measure. The three

Component	Values
`FunctionCallTracer`	Function Call Trace Statistics
`BTreeAccess`	Access to B-Tree Pages
`RTreeAccess`	Access to R-Tree Pages
`HybridAccess`	Access to Other Hybrid Index Structure Pages
`EntriesLogger`	Entries put to Secondary Inverted Index Structures

Table 5.1: AspectJ Components and their Values

`<...Access>` components count the number of accesses to the particular components. `BTreeAccess` measures the hits to either `BTreeInnerNode`s or `BTreeLeafNode`s together with qualified information about the B-Tree structure accessed (either initial or secondary inverted index). `RTreeAccess` performs the same functionality for the R-Tree whereas `HybridAccess` counts the grasps to all other pages affected. Thereafter they are written in structured form to *.csv-files which can be processed further by using statistical analysis tools. The `EntriesLogger` component lists all operations executed at secondary inverted index structures in CSV form to analyse the actions performed there.
The probably most important component for measurements here is the `FunctionCallTracer`. This component uses the functionality provided by AspectJ to get access to all functions executed during the insertions of the hybrid index. Therefore, each call inside the control flow of adding a certain instance to the hybrid index is traced and analysed deeply later on. To persist the data permanently, an XML structure was chosen used to represent the call stack of the functions. As time is the most important attribute here, for each function executed, time is measured before and after the call. These data portions are then

persisted serialised into an XML file to be processed later. All of the writing of the XML file is done asynchronously in order to affect the actual program flow as little as possible. However, this bears the risk that at the end of the test suite run there is still a lot to be written to the disk as some kind of queue remains waiting to be processed. Another, probably more harmful, effect in this case is that the amount of main memory also rises due to the large amount of calls which are still in the queue. This can be avoided by just providing more main memory to the respective process which can be done in the virtual environment where the tests are executed. Thus, more main memory may be supplied if needed dynamically. The XML files generated during the test runs must be processed further by additional tools in order to generate the paths inside the program which contribute the most to the time required for execution. Therefore, on the one hand, a well defined xml schema (see appendix A.1) exists which is produced by the test runs and may be processed by the tools described in subsection 5.1.4. Together with the tools and the XML function call tracing component an entire profiling tool for dynamic source code analysis is created that allows the deduction of paths which contribute negatively to the run time behaviour of the index structures.

```
<trace:call name="void de.fhhof.iisys.index.hybrid.bitlist.BitlistHybridIndex.add(Session, Row)">
  <trace:call name="MasterPage de.fhhof.iisys.index.hybrid.bitlist.BitlistHybridIndex.getMaster()">
    <trace:call name="Page de.fhhof.iisys.index.hybrid.util.Util.loadPage(IHaveStore, int)">
      <trace:call name="PageStore de.fhhof.iisys.index.hybrid.bitlist.BitlistHybridIndex.getStore()">
        <trace:time>0</trace:time>
      </trace:call>
      <trace:time>5044000</trace:time>
    </trace:call>
    <trace:time>14876000</trace:time>
  </trace:call>
  <trace:call name="de.fhhof.iisys.index.hybrid.documentheap.DocumentHeap(MasterPage, HybridIndex)">
    <trace:time>0</trace:time>
  </trace:call>
  <trace:call name="SearchRow de.fhhof.iisys.index.hybrid.bitlist.BitlistHybridIndex.getSearchRow(Row)">
    <trace:time>43000</trace:time>
  </trace:call>
  <trace:call name="List de.fhhof.iisys.index.hybrid.bitlist.BitlistHybridIndex.getWords(SearchRow)">
    <trace:call name="List de.fhhof.iisys.index.hybrid.bitlist.BitlistHybridIndexDocumentOpclass.getWords(Value)">
      <trace:call name="Object de.fhhof.iisys.index.hybrid.util.Util.deserialize(byte[])">
        <trace:time>451000</trace:time>
      </trace:call>
      <trace:call name="List de.fhhof.iisys.index.hybrid.bitlist.BitlistHybridIndexDocumentOpclass.getValueList(List)">
        <trace:call name="byte[] de.fhhof.iisys.index.hybrid.util.Util.serialize(Object)">
          <trace:time>48000</trace:time>
        </trace:call>
        <trace:time>943000</trace:time>
      </trace:call>
      <trace:time>3242000</trace:time>
    </trace:call>
    <trace:time>4157000</trace:time>
  </trace:call>
```

Figure 5.3: Excerpt of a Call Trace

An example of a call trace satisfying the given xml schema can be seen in figure 5.3. The function calls are organised hierarchically in order to reproduce the stack trace of a function properly. The main attribute of a call is the function call signature. Additionally, there is an element displaying the time needed for the execution of the particular function execution. It must be noted that the time in this case contains all subsequently done function calls carried out by the particular function. This technique enables to generate all paths of the program dynamically and also to extract the hot spots where most time is necessary inside the program execution.

5.1.3 Test Data

In order to measure the attributes of the hybrid index structures, a well-defined data input is required, as well. Therefore, freely available and representative data sources must be found which can serve as input data for the hybrid index structures. As the main target for the existing hybrid index structure managing geographical coordinates and full texts simultaneously is to index these heterogeneous data types, also data sets which contain these types of data must be found in order to generate representative data input values.
At least two data corpora must be chosen to avoid fitting the results of the optimisation towards one particular data input. Therefore, one corpus can be chosen as test input and the other one

as verification corpus. Mainly, this index structure was designated to support data produced by a search engine crawling for news articles. This search engine is called SARA2 [64] and was developed in a project from 2008 to 2010. For this reason, a news article corpus must be found with similar properties as the data produced by the search engine. It was decided to take the Reuters corpora, which are freely available for research purposes, for such kind of input (see subsection 5.1.3.3).
On the contrary, the index structure should not be limited to news items. That is why, another corpus must be selected which is more generalised than the news articles from the Reuters dump. The articles from Wikipedia are selected in this case because the data dumps are also freely available.
The third kind of input is a synthetic data set. It may be produced by using a data generator which creates documents based on statistical properties. This data generator is helpful for producing certain kinds of data to reproduce specific effects occurring inside the data sets. Therefore, it may serve to develop input data following some statistical laws to show certain effects inside the database.
The characterisation and the generation process of the input data are explained in the following subsections.

5.1.3.1 Processing of the Corpora

The test data to be used as input should contain geographical coordinates as well as a full text part. So, appropriate methods must be selected to extract the desired types of data from the texts. As the amount of data is very high (several GB to process), an accommodated method for extracting the data must be chosen. An Apache Hadoop[3] cluster was set up to extract the data efficiently.

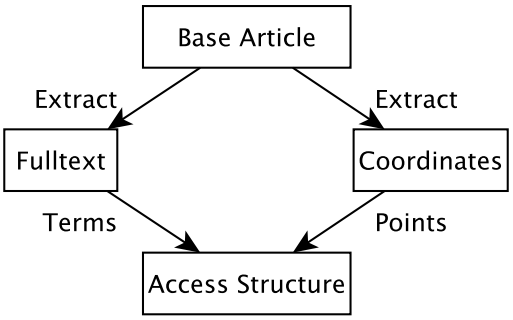


Figure 5.4: Extraction Process

An overview about the extraction process can be seen in figure 5.4. The general execution is to split the extraction in two parts. First, the full text is split into a set of terms. This is achieved by the use of the Apache Lucene[4] library. It already provides all the analyses required to split the full text into a list of terms, remove stop words, perform character normalisation, etc. Thus, nothing needs to be added in terms of functionality to extract the pre-processed terms which are fed into the index structure afterwards.
The part of extracting geographical coordinates results uses the same components as applied in the SARA2 search engine. Therefore, this is a well-known method for extracting first toponyms from an unstructured or semi-structured text and afterwards assigning geographical coordinates to these previously extracted place names. Besides this extraction, another analysis is carried out when using the Wikipedia dump.

[3] `http://hadoop.apache.org/`, accessed 2013-07-05
[4] `http://lucene.apache.org/core/`, accessed 2013-07-05

5.1.3.2 Wikipedia Dump

The Wikipedia dump is generated repeatedly from the existing Wikipedia pages. A database dump is done and converted to certain XML based representations using MediaWiki syntax. For a detailed XML schema see `http://www.mediawiki.org/xml/export-0.7.xsd`[5]. The dump used as data input is generated from 2011-10-07. Unfortunately, the dumps are deleted after a certain period of time based on disk space requirements. The original dump used the pages-articles excerpt which could be accessed at `http://dumps.wikimedia.org/enwiki/20111007/enwiki-20111007-pages-articles.xml.bz2` unfortunately is already deleted. Current dumps can be downloaded at `http://dumps.wikimedia.org/enwiki/latest/`[6]. The analysis methods provided to the dump, however, are also still working for newer data extractions. Yet, the data to be used for testing had to be chosen at the beginning and so it does not make sense to generate a new set of documents today.
Besides the analysis methods, described in the previous subsection, Wikipedia also provides articles or contents augmented with geographical markup. Therefore, in some cases, previously annotated coordinate values of points of interest inside articles can be extracted from the texts. This also means that no kind of heuristic must be used to get the actual coordinate values.
The basic values of importance for textual analysis in this case are Zipf's Law [106] and Heaps' Law [44]. These are two empirical laws of special importance for information retrieval. Zipf's Law describes the distribution of elements, e.g. terms in texts, to the entities, e.g. textual documents, they are contained in. In many cases, this power law distribution exists.

$$z(rank) = \frac{c}{rank^{\alpha}} \tag{5.1}$$

The basic model of Zipf's Law is given in equation 5.1. This law states the frequency of a particular element inside a collection whereas the elements are ordered descending based on their occurrence. The parameter *rank* is the rank of the element inside the ordered collection. c and α are corpus specific parameters which must be derived from the set of documents at the beginning.
Heaps' Law describes the growth of the individual elements with respect to the amount of total elements.

$$v(q) = K \cdot q^{\beta} \tag{5.2}$$

The equation 5.2 formalises Heaps' Law. The individual items $v(q)$ are calculated from the total number of elements q and the parameters K and β. These two parameters are corpus specific parameters and thus subject of investigation for the test input data.
Zipf's Law and Heaps' Law are some of the most fundamental statistical laws for information retrieval related analysis. Therefore, they are analysed for the text corpora used here. As already noticed (e.g. [73]), these two statistical laws are also closely related to each other.
Both laws are derived from the previously analysed text corpora. That means that stemming, normalisation and point extraction have already taken place. For this reason, two value sets can be extracted for both, texts and points, the laws are calculated from.
The Zipf's Law distribution of the Wikipedia data regarding the terms can be seen in figure 5.5. The parameter α, here displayed as $a \approx 1.18$ and $c \approx 7.06 \cdot 10^7$. The values to fit to the curve are calculated from the measured data using linear regression.
Heaps' Law values are given in figure 5.6. In this case c is the corpus specific parameter K and a is β. The value for K is $K \approx 4.34$ whereas β is $\beta \approx 0.711$ which means that the growth of individual elements regarding the absolute number of elements q is approximately $q^{\frac{3}{4}}$.
The Zipf's Law plot of the Wikipedia data for points (see figure 5.7) contains a constant of $c = 3.5 \cdot 10^5$ and an exponent of $\alpha = 1.038$. Based on these given values, it can be seen that Zipf's Law also applies here as the exponent conforms to valid values.
The Heaps' Law data for the points of the Wikipedia corpus are shown in figure 5.8. The constant value of K is $K = 1.36$ in this case and the exponent $\beta = 0.786$. That means that the amount of new elements regarding the total number of points inside this collection grows with an exponent of $\approx q^{\frac{3}{4}}$.

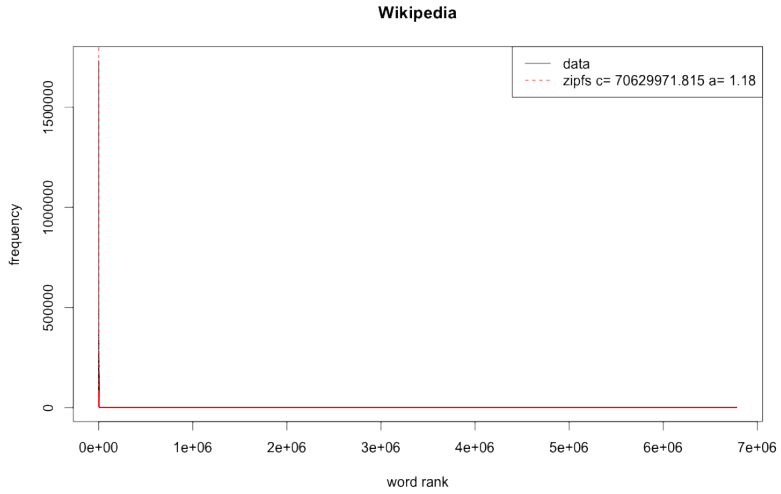


Figure 5.5: Zipf's Law Plot of Wikipedia Dump

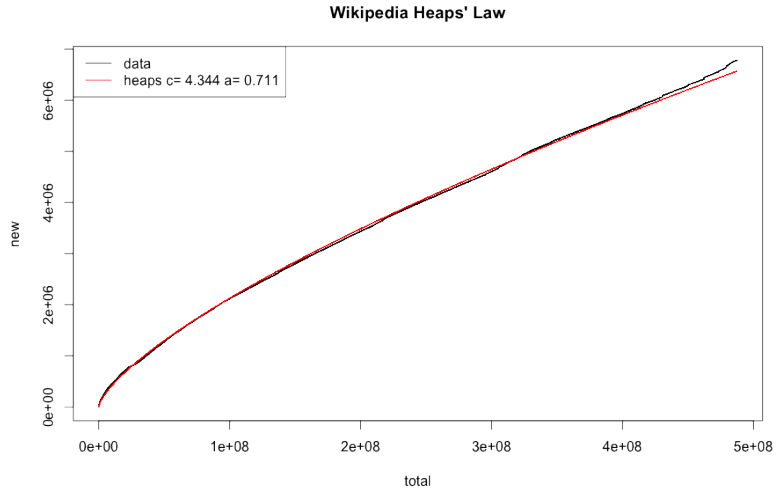


Figure 5.6: Heaps' Law Plot of Wikipedia Dump

With respect to the amount of elements inside the respective data collection, statistical statements can be made based on the following figures.
Figure 5.9 displays the frequencies of text lengths inside the Wikipedia corpus. The minimum number of words inside the Wikipedia corpus is one as empty documents do not make any sense to be analysed. The maximum value for a distinct number of words inside a Wikipedia article contained in this corpus is 21,532. This is a very large article which also has a large number of points assigned. The median value for the number of words per article is 88 and the average is 170, which means that, generally, in terms of central tendency the Wikipedia corpus has a moderate amount of words per article.
The number of points per article is given in figure 5.10. The mean number of points is 2.67 whereas the median value is 1. The minimum amount of points per article is 1, the maximum is 775.
These values describe the general statistical values found inside the Wikipedia corpus. It must be noted that subsets of this article collection differ a lot. However, these are the main values collected regarding the entire set of articles.

[5]accessed 2013-07-05
[6]accessed 2013-07-05

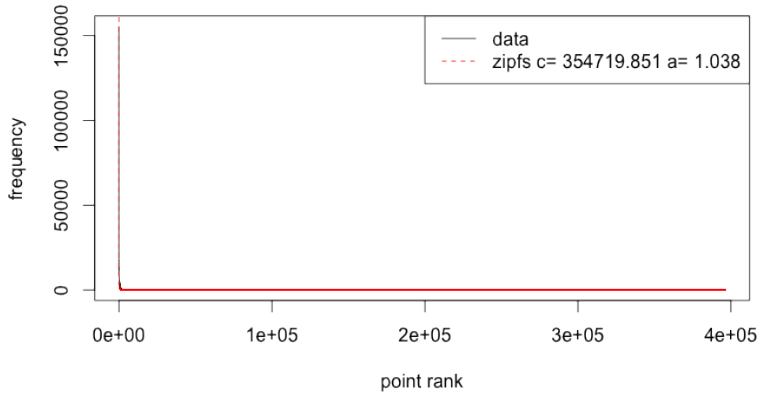


Figure 5.7: Zipf's Law Plot of Wikipedia Dump for Points

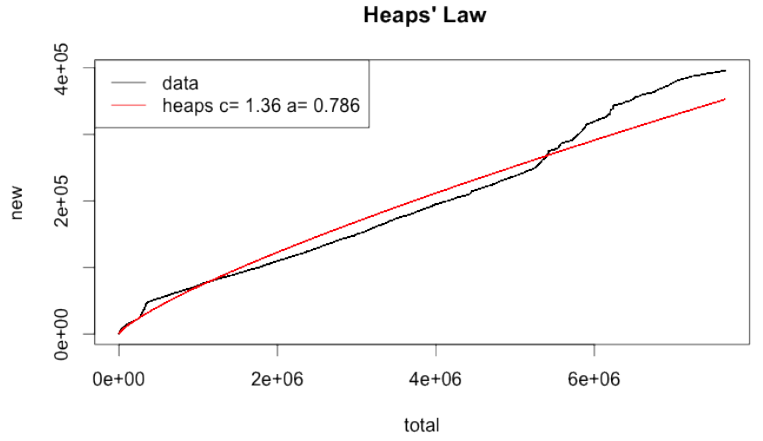


Figure 5.8: Heaps' Law Plot of Wikipedia Dump for Points

5.1.3.3 Reuters Collection

The Reuters collection is mainly also analysed to derive Zipf's and Heaps' Law values similar to the Wikipedia dump presented in subsection 5.1.3.2. The Reuters data used originate from the collection of data provided by the National Institute of Standards and Technology (NIST)[7]. It is a text collection of news articles supplied by the Reuters news agency. It is comprised by three different collections:

1. Reuters Corpus, Volume 1 (RCV1) in English language from 1996-08-20 to 1997-08-19 ($\approx 810,000$ articles),
2. Reuters Corpus, Volume 2 (RCV2) in multiple languages from 1996-08-20 to 1997-08-19 ($\approx 487,000$ articles), and
3. Thomson Reuters Text Research Collection (TRC2) in English language from 2008-01-01 to 2009-02-28 ($\approx 1,800,370$ articles)

[7] `http://trec.nist.gov/data/reuters/reuters.html`, accessed 2013-07-12

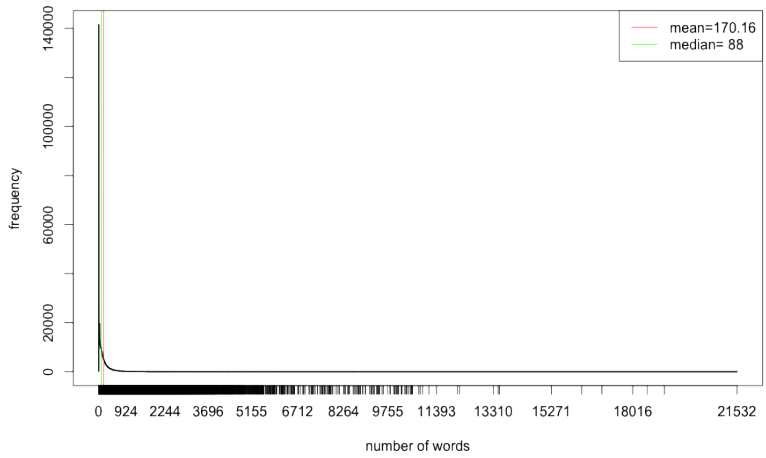


Figure 5.9: Number of Words Inside Wikipedia Documents

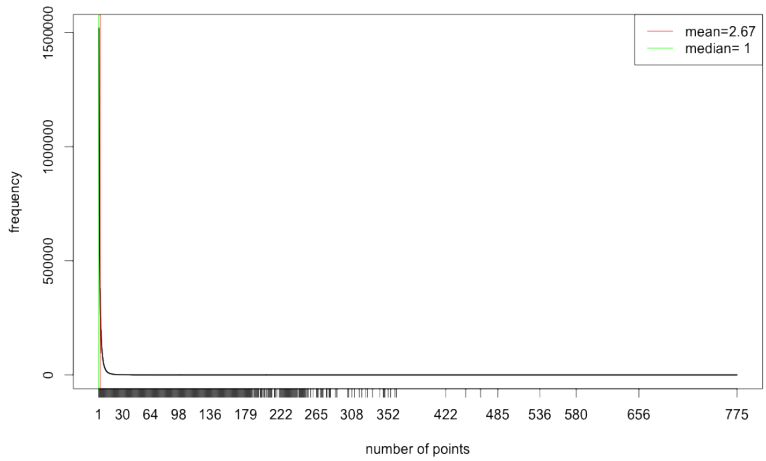


Figure 5.10: Number of Points Inside Wikipedia Documents

TRC2 corpus is one of the most frequently used article collections in text analysis research. All three of them are freely applicable for scientific purposes and thus, on the one hand, important for general text analysis research and, on the other, also available for this project. The decision taken here was to use the TRC2 corpus as it comprises most of the articles. The same process to extract the values from the articles (texts and points) is carried out and also the same kinds of statistics, mainly Zipf's and Heaps' Law, are extracted from the corpus data.
The Zipf's Law plot over the entire corpus collection can be seen in figure 5.11. The constant value c for the TRC2 data is $c \approx 1.278 \cdot 10^{10}$ and the α value is $\alpha \approx 1.926$. Comparing the α values of Wikipedia and TRC2 means that the relative frequencies of terms in TRC2 lower faster than in the Wikipedia corpus.
The Heaps' Law plot of the TRC2 data (see figure 5.12) produces a strangely looking curve as it does not rise steadily. This is, however, explainable because with the utmost probability the articles in the respective areas concern around one certain topic. Therefore, the vocabulary of individual words (to be seen at the ordinate) grows only slightly whereas the total amount of terms still grows linearly in this case. That is why the "leaps" result from a linear increase of total vocabulary with the individual terms only growing slightly. The values for K and β are therefore $K \approx 0.285$ and $\beta = 0.753$, respectively. Thus, the exponent β from the regression line

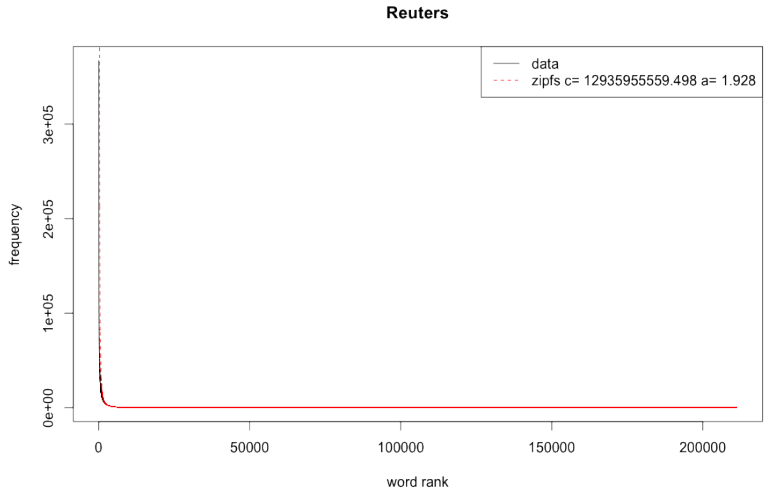


Figure 5.11: Zipf's Law Plot of Reuters TRC2 Data

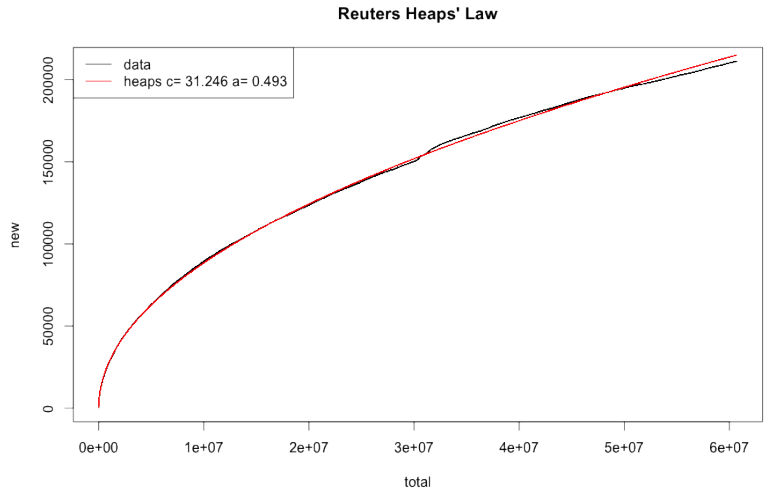


Figure 5.12: Heaps' Law Plot of Reuters TRC2 Data

is only slightly different from the exponent of the Wikipedia data, which means that both vocabularies grow similarly.
The Zipf's and Heaps' Law graphs for the points of the Reuters dump can be depicted from figures 5.13 and 5.14.
The Zipf's Law values for the Reuters TRC2 corpus for the points can be seen in figure 5.13. The constant value is $c = 1.3 \cdot 10^{10}$. The exponent is $\alpha = 1.928$. Compared to the Wikipedia data, the exponent is significantly greater, which results in a faster decrease of frequency comparing the ranks of the items.
The Heaps' Law plot for points inside the Reuters TRC2 collection is shown in figure 5.14. The respective values are $K = 31.246$ and $\beta = 0.493$ which means that the growth of new items to the total quantity of items is $\approx \sqrt{t}$.
A summarising figure about the number of words inside Reuters corpus documents can be seen in figure 5.15. It displays an average number of 144.26 words per article and a median value of 149. The minimum amount of words inside one article is 1 whereas the maximum is 321 different unique words per article. These quantities show that the range of values is much narrower than the one given in the Wikipedia article collection.
The frequency of the number of points inside Reuters documents is given in figure 5.16. It shows a median value of 1 and a mean value of 1.73. The average, in this case, is lower than the

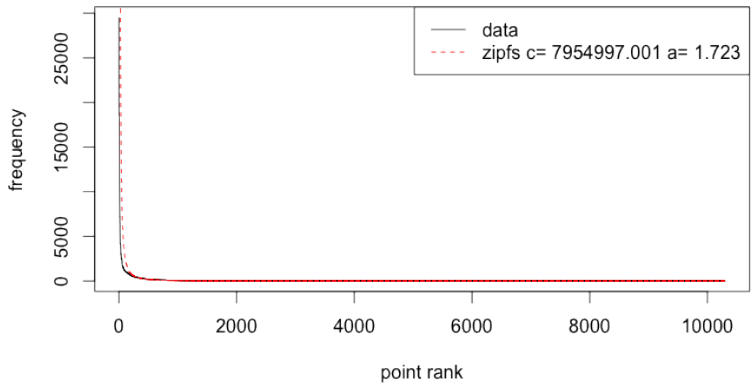


Figure 5.13: Zipf's Law Plot of Reuters TRC2 Data for Points

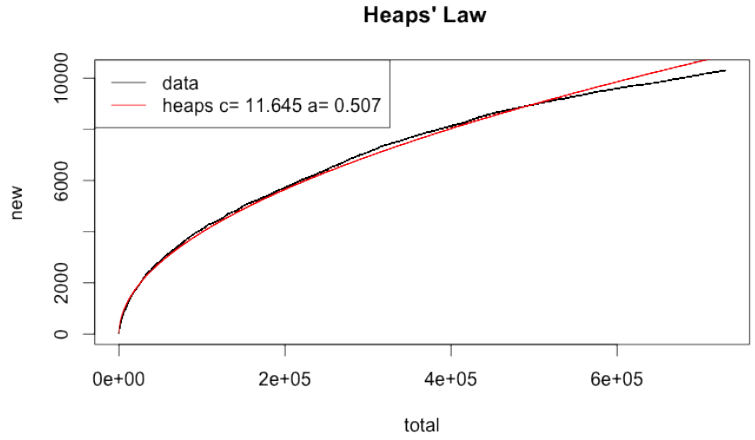


Figure 5.14: Heaps' Law Plot of Reuters TRC2 Data for Points

one given in the Wikipedia document collection. Additionally, the minimum value is 1 and the maximum number of points inside an article is 76 which again results in a narrower range for the possible values.

5.1.3.4 Data Generator and Synthetic Data

Besides the two real world data sets obtained from Wikipedia and Reuters a synthetic data generator is also created. It is basically present to create data which resemble the structure of present excerpts of corpora data. The generated values can then be used to fill data into the database in order to recreate certain circumstances and show specific properties of structures under test.
The data generator is mainly created to fit to values of predefined Heaps' and Zipf's Law properties. Therefore, generators must be implemented which enable to fit the data to these statistical laws.
Besides creating the document corpus according to certain freely choosable values, presets like data taken from Wikipedia or Reuters may also be applied.
Figure 5.17 shows a screenshot of the document generator tool. It is also feasible to generate only texts and no points and to specify the number of documents to create. For both types of

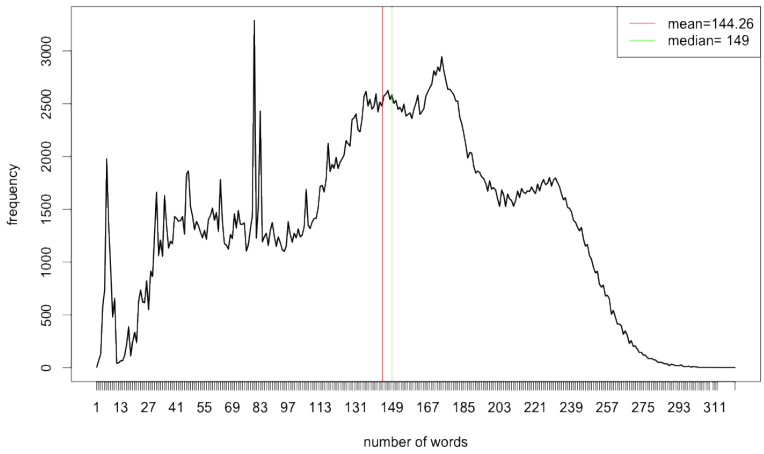


Figure 5.15: Number of Words Inside Reuters Documents

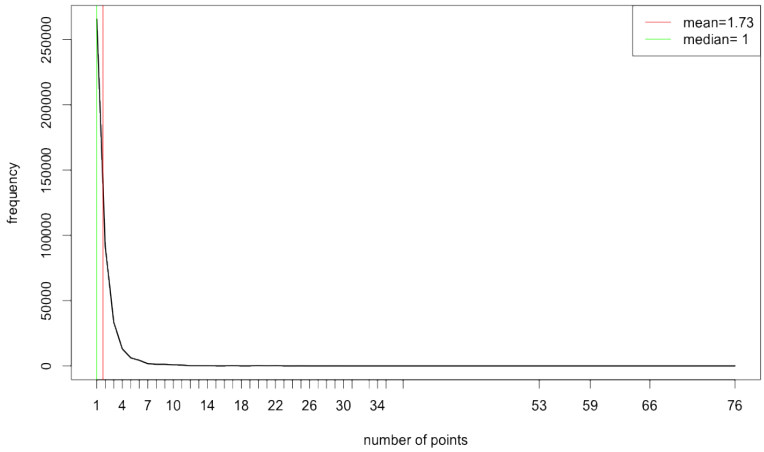


Figure 5.16: Number of Points Inside Reuters Documents

data, texts and points, it is possible to generate Zipf's Law and Heaps' Law constants independently and to fit the data properly to these types of data.
The document generator produces no real data sets. Therefore, the words which are created are not taken from a list of allowed words which are afterwards distributed to the documents. On the contrary, artificial words are created just by the use of random generators in order to solely fit to the statistical laws.
The schematic process of text generation can be seen in figure 5.18. First, the lengths of the texts are generated which are also freely definable. After that, a certain component generates random words, which means that just sets of individual printable characters are produced and combined in random order. That means that the words as such do not have any meaning in whatever language. Due to the randomness there is a certain likelihood that some of the words have a meaning which is, generally, not intended. Having created the amount of individual words which are based on the text lengths and the properties of Heaps' Law, a Zipf's generator is used to produce assignments of terms to an amount of documents in which they are supposed to occur. Therefore, both laws, which are also closely related, can be fulfilled by the data set. Having created the assignment, the text generator thereafter continues to assign the terms to documents by using the frequencies instantiated by the Zipf's generator. However, the Heaps'

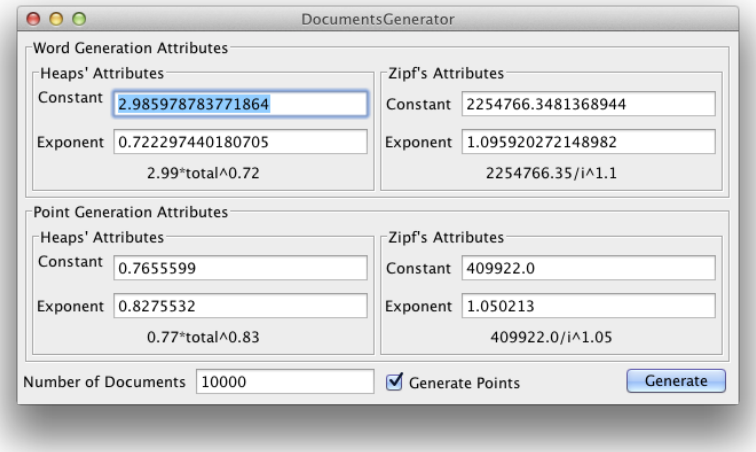


Figure 5.17: Screenshot of the Document Generator Tool

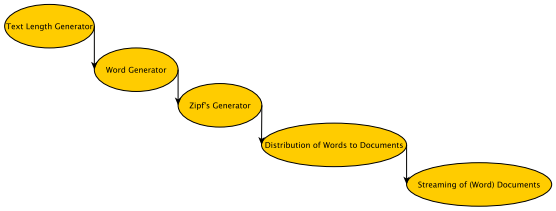


Figure 5.18: Schematic Process of Text Generation

Law statistics are also kept here to fulfill the requirements regarding the growth of the individual vocabulary of the document corpus. After having created this assignment an exporter writes the textual documents to the disk (first in serialised form) to process them further later on.

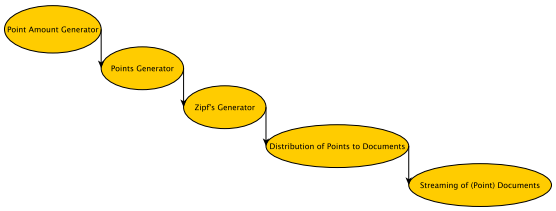


Figure 5.19: Schematic Process of Point Generation

The point generation (see figure 5.19) is done in a similar way as the text generation. However, the distribution of the points may not be as arbitrary as the purely randomly generated sets of characters for the words. Therefore, a steering mechanism is implemented to assign points to certain clusters. In general, this is inspired by real world data sets where the point values are distributed over the continents. In seas there are mainly not any points of interest (at least in most cases). For this reason, the frequency of spatial references in texts is much higher on continents.
A geographical coordinate cluster generator is used to create point values inside a certain region. The size of the region is also used to depict the relative frequency of occurrences inside. A screenshot of the user interface component responsible for generating statistically meaningful point values in a finite domain is shown in figure 5.20. The existing continents of the earth, in roughly approximated form, are taken as default values here. Yet, the distribution of the clusters as well as their size and position may be chosen arbitrarily. However, for generating generalised

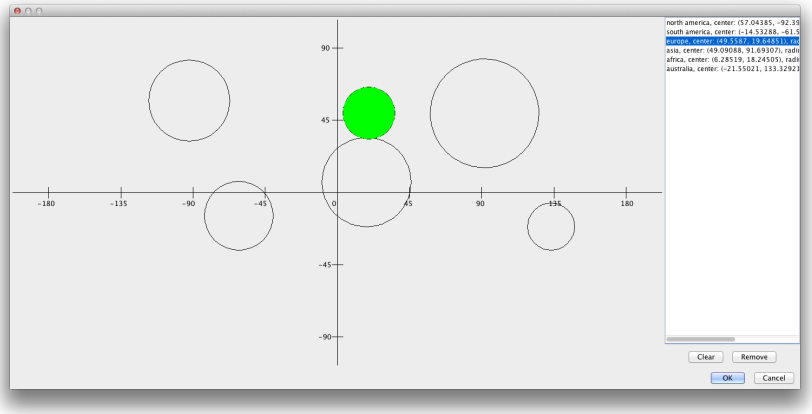


Figure 5.20: Screenshot of the Cluster Generator Tool for Points

document sets, it probably makes sense to generate values which are not too far away from existing properties. As already stated, the amount, size and position of the clusters may be chosen arbitrarily. The only thing which may not be chosen arbitrarily is the shape of the clusters. They are always circles. This is just a property inspired by design and calculation. Inside the clusters, two-dimensional Gaussian distributions are used to generate the points. More sophisticated properties could be selected here. However, this data generator solely serves for generating approximate values to show specific concepts and effects. Therefore, in most cases, it is probably sufficient to generate the points using two-dimensional Gaussian distribution functions.
After the generation of the points is finished they are also written to the hard disk to wait either for further processing or the final export of the point data.

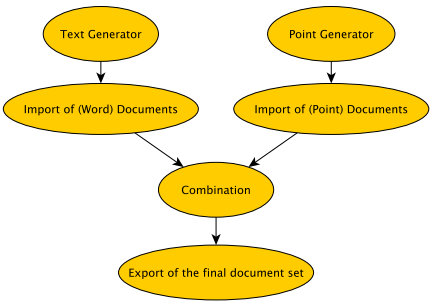


Figure 5.21: Schematic Process of Document Generation

Figure 5.21 shows the entire process of the final document generation. Here, texts and points can be generated in parallel in different threads. As the generation of the texts, normally, takes longer than the generation of the point values, the cluster generator view is opened after the document generation process is initiated. Thus, the text generation may already start while the clusters of points are being created. After both of the respective generation processes are finished, the combination of the two features may take place to generate the final set of documents.
First, both types of features are reloaded from the exported files the particular generators exported the parts to and then they are combined with each other. Some statistical generators may also apply here in order to, e.g., create a valid assignment of an amount of terms to an amount of points inside a document. When this combination process is finished, the final set of documents is created and may be exported to the disk. These documents thereafter fulfill the statistical properties assigned in both affected value domains of texts and points in the beginning.

5.1.4 Analysis and Tools

Besides the tools, described in the previous subsections, also further analysis methods exist which may handle the data generated in the particular phases. This implies that, on the one hand, measurement techniques must exist which analyse the particular statistical laws. On the other, tools analysing the data generated by the test suite described in advance must be disposable, too.
The main statistics are derived by using software for statistical computing. In this case, R[8] was chosen because it is freely available and open source software. As, besides the measurements about the documents and their statistical properties, there also are a lot of numerical values about, e.g., the amount of loaded nodes inside the hybrid index structure, these values can be easily processed by using R.
However, some of the tools, especially for analysis tasks, should be mentioned here, as well. The main target for this work is to improve the runtime behaviour of a hybrid index structure running inside a relational database engine implemented in Java. The given measurement techniques and types of exported data are already described in subsection 5.1.2. But these techniques only refer to the collection of data. As regarding the runtime of the algorithms the main focus lies on evaluating the XML dumps generated during the particular test runs executed by the test suite. Therefore, it is important to work on the exported XML dumps further. This subsection describes the process to import the particularly exported data again and to process them in order to get hints for additional static code analysis methods which paths or functions or algorithms in the entire process of inserting data into the database to optimise. Therefore, two basic methods for processing the data further are described in this subsection.

5.1.4.1 XML Importer

The first step for working with the extracted data of the program in XML form is the simplification and summarization of the runtime of the algorithms inside the program. Only one particular measurement does not make sense for representative results. Therefore, the following analysis works on summarised values. That means that the particular functions and procedures inside the index structure must be represented in a compact way.
The decision was made to work on either maximum or average values. For this reason, the cumulative sum of the runtime of the respective functions must be built and also the amount of calls to one particular procedure will be logged. Another requirement for the importer tools is that it must create a sufficiently small (in terms of disk space) representation of the data collected from the program. In some cases, the creation of the XML dumps generates files with large sizes. This even reaches terabyte regions. Hence, a more compact representation of the data must be found, which makes it easy to, on the one hand, move the files to certain places (copy) and, on the other, to be able to analyse and process them fast in order to get meaningful results. A lot of profiling tools also use a representation of the total amount of time spent inside one particular procedure and the number of calls to this function.
A practicably reasonable description of function call sequences inside programs are paths of program calls which occur in sequences. The paths inside the programs may interconnect functions of various types. This leads to graphs of calls as paths from functions may be interconnected. Hence, a sufficiently meaningful description of these paths must be found in order to analyse them properly.
The Neo4j[9] database is a graph oriented database making it easy to store the paths derived from the programs, efficiently.
The schema chosen for storing calls in the Neo4j database is shown in figure 5.22. The calls with given names point to other calls (or may, in case of recursion even call themselves) augmented with a certain time and an amount. These two features are the main attributes of interest for the analysis later on. The importing process uses XML streaming to pick data from

[8] `http://www.r-project.org/`, accessed 2013-07-12
[9] `http://www.neo4j.org/`, accessed 2013-07-12

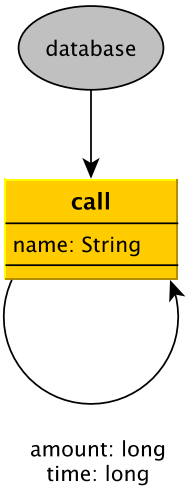


Figure 5.22: Schema for Saving Calls in Neo4j

from the XML files because these files are likely to be large. Whilst reading the data from the XML dumps, the particular function call nodes in Neo4j are created and interconnected to represent the paths of the program properly. The time and amount properties of the particular edges are also modified to represent the extracted values properly.
This process produces a Neo4j database containing all the measurement information of one test run. This information is, however, still raw and only pre-processed in terms of summarization. Consequently, at this stage, no information can be given which paths or algorithms to optimise, later on. The import only facilitates summarising the results given in the potentially huge XML files generated during the measurements. Therefore, a tool is needed with the ability to work with the existing paths imported into the Neo4j database in order to process the data further.

5.1.4.2 Paths Analyser

As already described in subsection 5.1.4.1, the XML Importer tool solely imports the XML dumps generated by using the test suite and summarises the information. This summarization is done by performing a lookup on the given function or procedure name and, if present, modifying the amount and total time required for the call. This step looses a little bit of accuracy as it does not allow keeping particular call times but summarises them by providing the total time of all calls. Using the amount of calls which is also kept during imports (see figure 5.22), an average time for a certain call may easily be derived.
The total accuracy is also not required for the analyser tool as it only serves for presenting hints to the user. Therefore, the main target for the paths analyser is to present certain procedures or algorithms which do not run efficiently inside the entire software. Additionally, further inspection using static code analysis approaches is needed.
The system may not solely suggest certain parts inside the algorithms to be optimised. Based on the fact that, hypothetically, software parts might exist which have already been optimised or are too short to be optimised in some cases it is not possible to improve them. For this reason, the paths analyser tool simply proposes the parts of the program to inspect further, either with static or further dynamic code analysis techniques.
The paths analyser tool has the ability to import data from a Neo4j database generated by using the XML importer. That is why, it relies on the same schema as described in subsection 5.1.4.1. The target of the paths analyser is to outline those paths which contribute most negatively to the entire runtime of a program, or parts of it. The tool recreates the individual paths visited during the run of the program by traversing the graph database in a predefined manner. Neo4j

already introduces a technique to achieve this using the "Traversal Framework" (see [97, p. 53 – 61]). The traversal framework is configured to use a depth first search approach, path uniqueness in order to include each call node in each path only once and sort the call nodes according to the total duration. It uses three customised settings to generate the call tree from the graph stored inside the Neo4j database:

1. An expander which follows relations based on certain predicates
2. A node filter with the ability to prune paths
3. A uniqueness setting defining the cardinality of a certain node

The node expander defines which relations to expand in the next step. Therefore, sorting is already applied here to prefer longer paths (in terms of duration). Additionally, only outgoing relations are expanded in order to prevent loops at this stage. The avoiding of loops is additionally supported by the uniqueness setting applied. It is adjusted to a node path basis. Therefore, each node may appear inside one particular path only once. This additionally tries to eliminate potentially occurring loops inside the evaluation. The node filter configured here has the ability to prune entire paths. They include getter or setter functions which will probably not much affect the entire runtime.

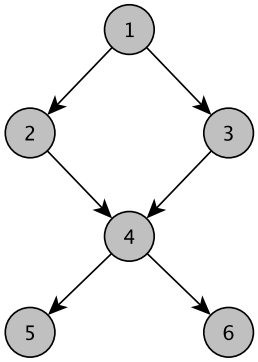


Figure 5.23: Example Call Graph

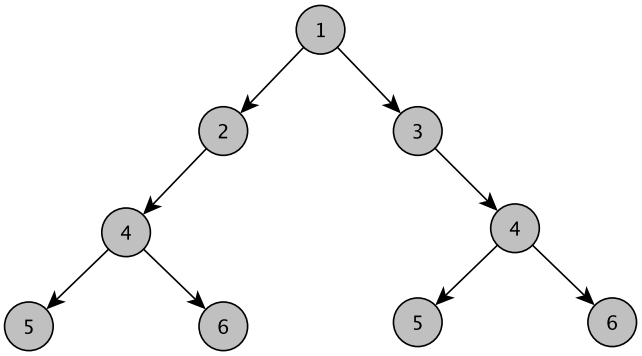


Figure 5.24: Example Call Graph Converted to Call Tree

An example for a call graph is shown in figure 5.23. This call graph is supposed to be converted into a call tree using the traversal framework. Based on the settings applied to the framework, the traverser generates the following individual path elements:

- $1 \rightarrow 2 \rightarrow 4 \rightarrow 5$
- $1 \rightarrow 2 \rightarrow 4 \rightarrow 6$
- $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$
- $1 \rightarrow 3 \rightarrow 4 \rightarrow 6$

These are the paths which fulfill the given options due to the settings of the expander, node filter as well as uniqueness in a depth first search manner. It can be seen that this path list can easily be converted into a call tree by using a common sequence of subpaths. The outcome of the algorithm which converts the call graph into a call tree applying the adopted version of the traversal framework is displayed in figure 5.24.
This generated call tree resembles the structure and all paths of the original program in a summarised form whereas the particular call paths are sorted in a descending way, which means that those paths contributing most to the runtime of the algorithms are already sorted to

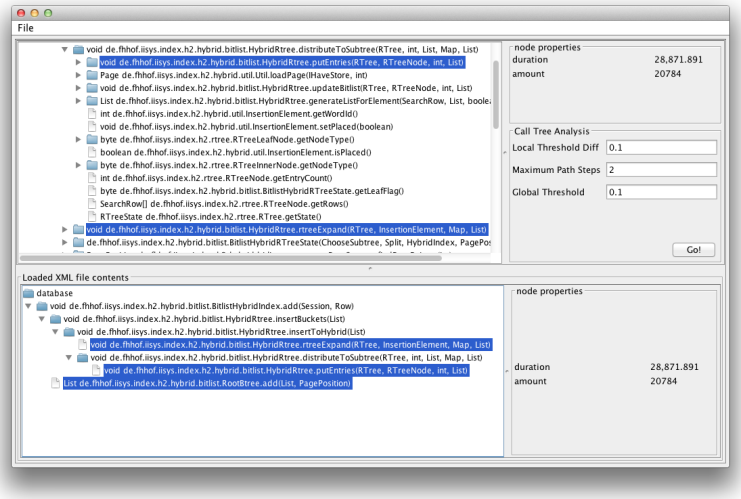


Figure 5.25: Screenshot of the Paths Analyser Tool

the top of the call tree representation. This call tree which finally results from the dynamic code analysis that uses the test suite generating the XML trace files which are imported into a Neo4j database using the XML importer tool is afterwards displayed in a graphical user interface to give the user the possibility to process the data further.
Illustration 5.25 shows a screenshot of the paths analyser tool. The call tree of the program is demonstrated in the upper part of the window. The lower part of the window displays exported paths from a previous iteration. These paths may be exported in a structured way and re-imported in order to carry out comparisons directly inside the program. The right half of the window shows the total duration and the number of calls to a specific function, respectively. These numbers are updated on click to a certain path. Therefore, this tool can already provide an overview of the given actions executed inside the program and the paths which contribute most to the runtime of a certain algorithm. However, there are still more sophisticated methods to extract specific parts of the program.

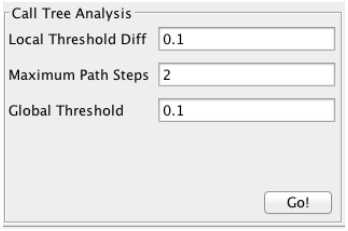


Figure 5.26: Parameter Selection in the Paths Analyser Tool

An excerpt of the screenshot is enlarged in figure 5.26. This part of the program is especially important for generating the hints indicating which paths or algorithm parts must be optimised during the following iteration.
As the goal of the tool is to provide just a small amount of relevant program paths, additional filter capabilities may be applied in order to generate only a few algorithms to be optimised potentially. In general, the paths analyser performs a depth first search on the given call tree to generate the optimisation hints. Three parameters may be modified in order to retrieve meaningful optimisation hints from the tool:

1. Global Threshold, referring to "uninteresting" paths (t_{global})
2. Local Threshold, meaning "hardly decidable" paths (t_{local})
3. Maximum Path Steps, referring to the maximum amount of steps to follow one single path

5.1.4.2.1 Global Threshold The global threshold value which can be set inside the paths analyser tool refers to a decision between "interesting" and "uninteresting" paths. This threshold can be set to prune paths inside the imported database which do not much contribute to the total runtime of the program. That means that this threshold directly removes paths which only take a small percentage of the "parental" call. The parental call, in this case, is the function which calls the currently inspected one. More formally, the global threshold only allows to keep calls whose duration positively validates according to equation 5.3.

$$t_{global} < \frac{duration_{call}}{duration_{parent}} \tag{5.3}$$

For this reason, those paths are pruned whose percentual duration regarding the parental call is smaller than the global threshold. This filter already has the ability to remove a lot of sub-paths which do not much contribute to the total runtime. Depending on the percentual setting of this threshold, it either removes a lot of or only few paths. However, as the goal of the paths analyser is to provide "a couple" of algorithms only, it is probably reasonable to set the threshold value to at least 50%. Yet, the settings may be changed arbitrarily in order to generate meaningful paths.

5.1.4.2.2 Local Threshold The local threshold value which can be set is present to stop the analysis at stages where the influence of multiple paths inside the program does not differ much. On account of this, the decision about which of the multiple paths have the greatest influence on the entire runtime is hard to take. Thus, the algorithm of analysing the call tree is stopped for the current branch and all calls which cannot be easily separated according to their runtime are returned. Formally, the local threshold can be represented corresponding to equation 5.4.

$$t_{local} > \left| \frac{duration_{call_1}}{duration_{parent}} - \frac{duration_{call_2}}{duration_{parent}} \right| \tag{5.4}$$

That means that paths are added to the list of optimisation candidates if the difference of the percentual duration of two or more calls regarding the parental call duration is lower than the local threshold. Therefore, by setting a high value to the local threshold nearly each of the paths satisfies the condition. Yet, limiting the local threshold to a low value implies that only paths are included which, on the one hand, need a high percentage of parental code and, on the other, do not much differ from each other. Setting the threshold to a low value thus makes more sense than setting it to a high value as only really "hardly decidable" paths will fulfill the criterion.

5.1.4.2.3 Maximum Path Steps The third parameter which can be set is the maximum path steps criterion. This is an artificial value to cut the path lengths in some cases. It also relates to the two thresholds described in advance. The basic meaning of this parameter is not to follow a certain path until the end. This could be possible if there was only one single path inside the procedure calls which has the most influence on the program runtime. Then the possibility arises that the suggested path to be optimised is a getter or setter function. Optimising at these stages is, however, not very reasonable because in most cases, getters just return specific values and setters assign them, respectively. The optimisation probability of these functions is quite low, the same holds for constructors in most cases or similar functions. Therefore, the maximum path steps criterion is introduced to stop searching an optimisation candidate too deep inside the call tree. Hence, some kind of counter is kept which states how many steps are done without considering any other path than the currently inspected one. If this counter reaches the pre-defined value of the maximum path steps parameter, the search is stopped and the current path is returned. This is also reasonable as, if no other path representing a sibling of the currently inspected one is searched, the current path obviously fulfils the global threshold parameter and in no any case fulfils the local threshold parameter. Consequently, it is obvious that this particular path contributes most, or at least much, to the total runtime of the program. Thus, this path is evidently an optimisation candidate.

When using these three parameters in combination with a depth first search inside the call tree, the optimisation candidate hints can be extracted by applying the paths analyser tool. However, as already stated, the tool cannot be used stand-alone. It does not take into account the concrete source code. Decisions are only made based on the pre-defined parameter values as well as the measured data generated by the test suite and pre-processed by the XML importer. Human inspection of the optimisation candidates is still required in order to determine the "optimizability" of certain code fragments, algorithms and functions. Certain validations may also be additionally carried out using other dynamic code analysis methods like a profiling tool.

5.1.5 Summary

This subsection explained the basic setup of the test suite used to carry out the tests. The values taken during the particular test runs stay the same over all iterations in the optimisation process. The test suite is applied to measure the current performance of the hybrid index structure. However, the basic functionality may be extended to a general purpose tool profiling an application to list the performance bottlenecks inside a Java application.
The analysis methods described in this subsection give basic hints on the worst parts of the program regarding the runtime. The values determined with the given tools, however, must still be verified with respect to the chances available to optimise certain program parts. Further techniques like static code analysis or the use of a profiler to measure specific parts inside the program are helpful tools to ensure the performance issues of the extracted parts. Static code analysis must be carried out in order to determine the disposable possibilities of rewriting certain code parts or restructuring the affected index access methods in order to speed up the code parts.
The tools specified here build the foundation of the work as they line out the currently existing performance issues. The measurement process used in the following subsections always has the same structure:

1. Carry out tests (pre-tests) and determine the performance bottlenecks,
2. List the solution alternatives, e.g. based on literature studies,
3. Select solutions to be implemented to overcome the currently existing weaknesses, and
4. Evaluate the chosen solutions by carrying out further tests (post-tests) and compare the results with the initial ones.

5.2 OPTIMISATION ITERATIONS

This section describes the particular optimisation iterations carried out during the actual research work. As a first step, a pre-test is executed which is run on the basic implementation as described in chapter 3. The general approach chosen here is to select a set of input documents from the pre-processed sets presented in section 5.1.3.
The fundamental database setup includes the definitions and extensions applied to the H2 database. There are two tables (`INDICES` and `OPCLASSES`) which form the basis for loading the index structures from external places. Besides these tables, an actual table used to insert the data in and query for the respectively stored portions of data exists.
The main table to construct the index on can be seen in table 5.2.

Table	Column	Type	Meaning
DOCUMENTS	ID	INT	primary key column using an automatic increment option assigning a serial increment of values
	WORDS	TXT	text structure containing the stemmed and normalised words in a string array representation
	POINTS	ARRAY	array of points representing the geographical coordinates found in the respectively analysed documents in the textual part
	DOC	DOCUMENT	computed column representing the combination of words and points

Table 5.2: Main Documents Table Structure

```
1 CREATE TABLE documents
2     (id INT PRIMARY KEY AUTO_INCREMENT NOT NULL,
3      words TXT,
4      points ARRAY,
5      doc DOCUMENT as make_document(words, points));
```

Listing 5.1: SQL DDL Statement to Create the Main Database Table

The SQL statement to create the main database table to set up the index on can be seen in listing 5.1. The document column is a computed one which is used to instantiate the particular hybrid index on. This column is built by uniting the words and points from one document row to a single document using a H2 specific extension. The computed column can be used to pre-calculate values from the respective other values.
The pre-computed column is used to set the index up on. Therefore, this is the main column which is employed for the tests. However, it is easier for the normal use inside a database system to utilise this way to insert data into the corresponding columns (words and points) and to apply the computed column for querying subsequently. This is also an easy way to query the particular columns individually and combine the search results afterwards in order to validate the search results.
An activity diagram of the actions taken inside the test suite is displayed in figure 5.27. This is the general execution process of the test suite. At the beginning, the parameters are read and the test suite is initialised. The particular measurement and export components are started and the tables as well as the index structures are instantiated if not already present. Subsequently, the actual testing process is executed. Then, the items potentially already existing are skipped in order to avoid adding duplicates. Thereafter, a pre-defined and configurable amount of elements is loaded into the database table. The amount of inserted elements is fixed to 25 during all the optimisation iterations. The next step in the execution is to perform a pre-defined amount of queries. They may also be read in advance so that a predefined set of queries may be specified for the test suite. After carrying out a particular number of these procedures, the test suite is stopped and the remaining data are collected and output to the disk. This is done in the last step of the test suite which closes all effected resources, finalises the data export components and closes the database properly.
The test suite may be configured to be run with the given aspects switched on or off. That means that this also allows the measurements. A result of preliminary test executions is that the amount of data gathered during the collection of the function call information is too large to run the test suite permanently with the respective aspects enabled. Therefore, the general process chosen for all subsequent iterations is to execute the test suite only at pre-defined measurement points with aspects switched on and off. The general process is to insert 50 documents with aspects disabled following one with aspects enabled continuing with 50. That

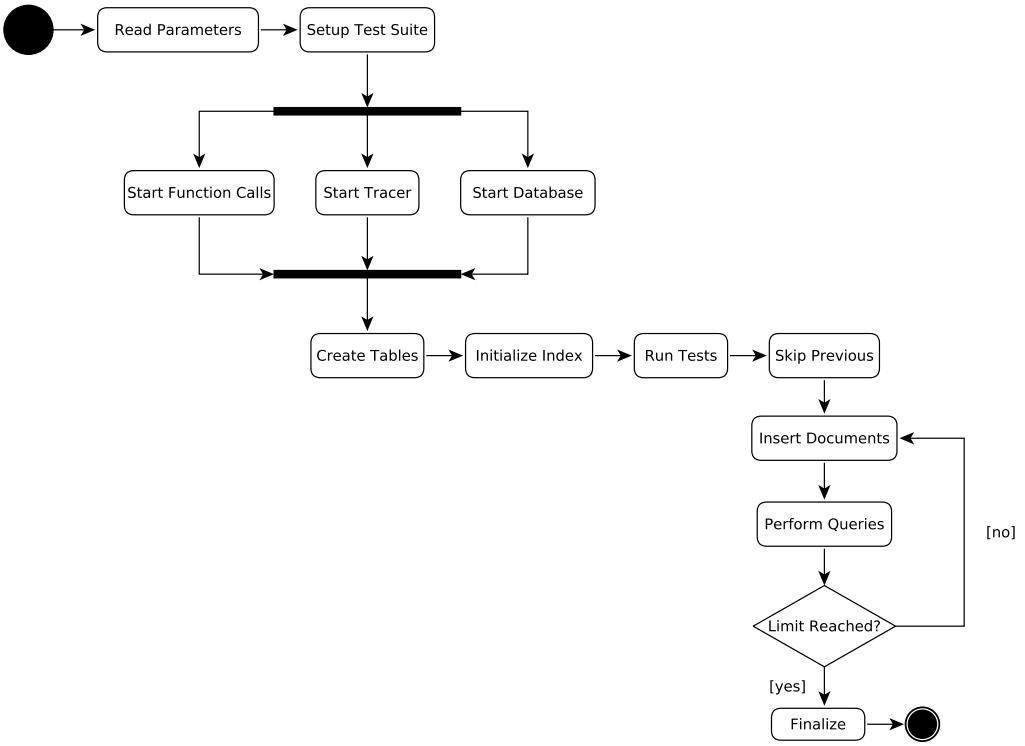


Figure 5.27: Activity Diagram of the Test Suite

indicates that after putting in 51 documents, there is a measurement point for the function call information. The remaining aspects stay activated permanently to gather also data from the queries if the function call trace procedure is disabled.

This section describes a series of optimisation iterations. As a first step, pre-tests are executed which lead to the first optimisation iteration. They produce output data to be analysed by using the respective tools specified in section 5.1.4. The calculated values then can be used as an input for searching solutions to overcome the currently existing weaknesses. It must be noted that the amount of elements inserted to the index structure is quite low at the beginning, because the performance of the initial implementation is very poor. Therefore, first, only a small number of elements inside the database under test exists. The amount of elements will be increased during the optimisation process as the time to put in the respective documents consequently lowers. More elements are inserted in each iteration to be able to verify the effects of the previous optimisation iterations also on higher amounts of elements. As a final step, a total measurement over all iterations is executed in order to reflect the entire optimisation process. This final measurement is then applied to a fixed number of documents to be able to compare the results of all iterations.

5.2.1 Pre-Tests

The pre-tests are executed as a primary step of the optimisation iteration in order to gather the first dataset. It then describes the initial state of the performance of the hybrid index structure regarding reorganisation efficiency. This test run also defines the basic test data to be inserted into the database.
The basic dataset chosen here is the one resulting from the analysis of the Wikipedia data. It is ordered by the article id assigned in the Wikipedia dataset. This id is already pre-defined by

Wikipedia and only re-used as id value, here. The ordering is an important property as the sequence of insertions then bases on one given in the article ids. That means that, on principle, the base dataset is the Wikipedia dataset. However, the particular effects of the optimizations must be validated by using another dataset in order to avoid optimising the hybrid index structure with regard to one particular document set. Therefore, the effects of the optimisation must be examined in respect of alternative input values. Hence, at least the Reuters dataset must also be used for verifying the actual results.
The testing environment consists of a virtual machine which employs a huge amount of main memory ($\approx$ 140 GB) because the intermediate storage of the results of the function call tracing analysis is stored in main memory. The actual memory consumption of the index structure is quite low. It has four virtual processors assigned to the machine. Each run is executed on exactly same machine to make a reproducibility of the testing results possible. However, the results should always be compared in relation to each other. So, it is only important to execute the test runs of two subsequent iterations on exactly the same machine.
The parameters for the database are fixed for each run as well. The database page size is set to 2048 bytes which is the default value for the H2 database. There are two additional parameters for the hybrid index structure to be set. On the one hand, the bitlist size must be set in advance and, on the other, the artificial upper bound *HLimit*. The bitlist size has an interdependency on the quantity of elements to be stored inside the R-Tree. Incrementing the amount of bits to be stored in one element lowers the number of hybrid R-Tree elements and vice versa. Therefore, the hybrid R-tree supports multiple ways to set this up. Either the bitlist size or the number of R-Tree elements may be specified as a parameter for the hybrid index. The decision is made to fix the amount of elements stored inside the R-Tree to five which means that five elements may be stored inside one R-Tree node at maximum. If the number of elements results in the storage of more than five elements, a split operation of the R-Tree is triggered which effects two new nodes. The split algorithms that can be chosen are described in section 3.2.3.2.
The second parameter to be set is the *HLimit* parameter. It decides whether a certain term is regarded as being frequent or infrequent. Only references to frequent terms are distributed through the hybrid access structure. Therefore, setting this parameter to a proper value is also a crucial task for the hybrid index. As no studies for setting the value for the hybrid access structure have existed so far, it is set to a fixed value of $HLimit = 200$. This indicates that a term occurring in more than 200 documents is regarded as a frequent one in the document collection.

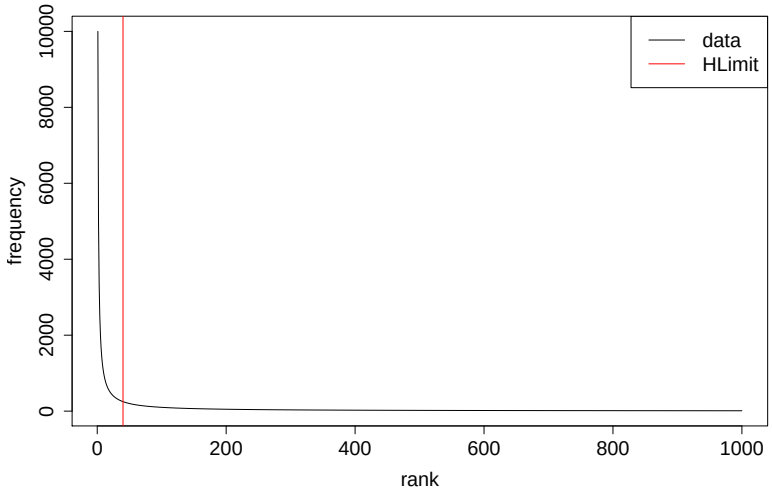


Figure 5.28: Conceptualisation of Zipf's Law / *HLimit*

A graphical conceptualisation of setting the limit value regarding a typical Zipf's Law plot is shown in figure 5.28. It can be easily seen that there is a large amount of terms (ordered via the frequency) appearing only very seldom, whereas only a small amount of terms occurs very often. That indicates that if the limit is properly chosen, solely the very small amount of very

frequent items is stored inside the hybrid part of the index. The remaining terms as well as the point references are stored in the inverted index (and document heap, respectively), only.
Generally, the pre-test run is executed to gather the first data without any preconditions. However, this first test run defines the circumstances for the remaining data as, basically, the fundamental definitions should preferably not be modified. The first test run is executed in 38 repetitions. As the basic approach is to insert 50 documents without any performance measurement, then one with performance measurement and so on, this leads to the fact that only half of the repetitions have a measurement of the function call traces attached. Thus, 19 measurements using the function call tracing aspect are executed in this run. The basic formula to calculate the number of documents n inserted in the respective database table for k repetitions is:

$$n = \left\lceil \frac{k}{2} \right\rceil \cdot 50 + \left\lfloor \frac{k}{2} \right\rfloor \tag{5.5}$$

The formula results from the fact that the first repetition is executed without tracing. For this reason, it must be noted that if k is odd, one additional run was executed without aspects. As the runs without aspects contribute 50 documents, the formula must be adapted to ceil the elements if k is odd.
Therefore, regarding 38 repetitions, there are in total $n = \lceil \frac{38}{2} \rceil \cdot 50 + \lfloor \frac{38}{2} \rfloor = 969$ documents in the database after the entire test run. This is not very much, however the first test run had to be stopped based on time constraints.
The test suite produces individual files per iteration. In the case of comma separated value (CSV) files, the test suite generates files using names with the repetition number attached. CSV files are applied for all data which may be processed in tabular form. That means that, e.g., the results of loaded nodes are stored in CSV representation. Hence, the first repetition produces files named "*.csv_0", the second one named "*.csv_1", ... The same procedure also holds for the XML files generated as call traces from the function call tracer component. This makes it easy to reconstruct the particular individual values for each rerun. Finally, it is also easy to combine the files in order to analyse the data from the entire iteration.
The results from this execution of the test suite are summarised in the subsequent iteration (see 5.2.2). This is done because they serve as an input for the analyses as well as the subsequent optimizations conducted during the iteration. The input data are then used for an analysis which paths to be optimised. Selected improvements are then first checked for the general possibility of applicableness and then included in the respective structures inside the hybrid index structure. As a final step, the post-test for each iteration is executed in order to validate the results inside the access method with the same configuration as in the initial test which leads to a check of the applied methods for validity inside the real index structure.

Parameter	**Value**	**Explanation**
HLimit	200	Artificial Upper Bound
Element Count	5	Amount of Elements per R-Tree node
Document Count	969	Amount of Documents to be inserted
Dataset	Wikipedia	Dataset to be processed (Wikipedia sorted by ids)
Split	R-Tree Split	R-Tree splitting method
Choose Subtree	R-Tree Choose Subtree	Method for Selecting a proper subtree to place an element in

Table 5.3: Settings for the Pre-Tests Test Suite Run

Table 5.3 summarises the relevant settings for the test suite and index configurations used for the pre-tests.

5.2.2 Page and Value Caches

Caching is known to bring advantages in databases and, more generally, in disk oriented systems. The principles of buffer replacement strategies, database caching and caching strategies have been part of research for a long time now (see i.e. [24, 25, 88]). Caches set up at certain places can be used to enhance the time to access a certain page or object. It is well-known that access times to pages can be lowered if a page replacement as well as a caching strategy are used which are adequate to the requirements of the respective algorithm utilising these pages. Later on, new versions of caches using multiple queues (see [55]) are also introduced in order to adapt caching algorithms to the requirements of loading database pages. This iteration tries to introduce cache related approaches in order to enhance operation speed of the reorganisations inside the hybrid index algorithms.

The first iteration results from the pre-tests. The evaluation of the testing results is always the same for each iteration. The experimental outcomes are evaluated regarding two different criteria. On the one hand, there is an evaluation for the different phases of the insertion process and, on the other, an iteration specific analysis method exists.
The phases evaluated for each particular iteration are related to the insertion process. This process first adds the rows to the initial inverted index. By means of this method a list of assignments of term ids representing bit numbers to lists of documents is produced. The term ids are assigned sequentially based on the insertion order. If one particular term overflows the artificial limit *HLimit*, a sequentially generated term id gets assigned to this term. This id is then reused on each following insertion operation. If an overflow of the limit occurs, either for the first time or repeatedly, exactly this term id is used for indicating an overflow at this stage. The term ids are assigned to a list of documents this term occurs in during the currently inspected insertion operation. These term to document assignments are then distributed throughout the hybrid R-Tree. Therefore, the first phase is the initial inverted index phase.
The second phase is the insertion of new items into the R-Tree to prepare the R-Tree for the succeeding distribution. That means that first the respectively required spatial structures are instantiated. This is done by the R-Tree phase which takes each point contained in at least one document of the overflowing list and inserts it into the R-Tree if it is not present yet. This phase utilises the R-Tree basic algorithms of choosing a proper subtree and a splitting method which may be supplied as an additional argument to the R-Tree. Thus, the second stage is the preparation of the spatial structures required to distribute all overflowing items to.
The third phase is the distribution phase. During this phase, the term to document assignments are distributed to the particular places where they fit spatially. That means that a term entry referencing a list of documents is distributed deeper into the R-Tree if at least one point contained in one of the documents of the list is enclosed in the spatial component of the R-Tree at the specific subtree. This procedure also determines the bits to be set inside the bitlist subsequently and is continued recursively up to the leaf level. There, the points inside a particular leaf node are compared with the points inside the respective document lists. This concludes the third stage.
The last step of the insertion algorithm is the placement to the secondary inverted index structures. Each R-Tree leaf element possesses a pointer to one particular secondary inverted index. Together with the point the secondary inverted index refers to, this index structure represents the final destination of the entries. Hence, the combination of the point and the secondary inverted index performs the final intersection step here. This index structure is constructed by the term id and the references to the document heap. As an additional step, the bitlists are reorganised. Each term id for each "valid" assignment is then entered into the bitlist in order to make the particular item refindable inside the entire structure.
For this reason, the particular phases analysed in the general analysis are:

1. Initial Inverted Index (Root B-Tree),
2. R-Tree Expansion,

3. Generation of Fitting Subelements,
4. Putting the Items to Secondary Inverted Index Structures, and
5. Entire Reorganisation Process.

Additionally, the particular values are compared to each other as a whole based on pre- and post-tests for each iteration in order to monitor the total results of the particular optimisation iteration.
Besides the general analysis, an iteration specific analysis is also carried out. The observation gathered from the execution of the test suite and a subsequent analysis of the results using the function call trace analysis tool is taken into account. Therefore, the respective evaluation methods described in section 5.1.4.2 are carried out in order to generate meaningful optimisation recommendations. That is why the path pruning of the paths analyser tool is substantially important as it serves as an input for the basic observations.
Each iteration section is structured as follows:

1. **Basic Observations** as results from the analysis including phases and iteration specific analysis. This subsection only lists the observations and does not provide any further information than the outcome of the empirical measurements.
2. **Solution Alternatives** for listing alternatives from literature studies which can be used for optimising the currently arising problems. These solution options give an overview of problem solving possibilities for the current issues.
3. **Selected Solutions** evaluates the solution alternatives, probably by supplying further test data as decision support, and selects the most promising solutions for inclusion in the hybrid index structure.
4. **Evaluation** subsection describes the evaluation by the use of post-tests. In this step, the post-tests are executed by using exactly the same settings as the post-tests. That means that the amount of documents to be inserted as well as the test suite and the outer parameters are the same. Yet, the outer parameters may also change in case they affect the performance and thus are a subject of change during this optimisation iteration.

5.2.2.1 Basic Observations

The basic observations for this iteration are shown in this subsection.

The total duration of the respective phases for iteration 1 can be seen in figure 5.29. The duration is displayed in millisecond resolution. However, the numbers may not be taken too strictly as the aspect oriented measurement approach adds additional time overhead to the existing time constraints. Therefore, the numbers do not exactly correspond to realistic time requirements as the measurement adds additional overhead here.
The first value inserted only for comparison is the add method representing the starting point of the functionality. It is easy to state that the dominant phase in this iteration is the generation of lists. This method consumes 83.59% of the entire time of the reorganisation process. The remaining methods only contribute the remaining $\approx$ 16.41% of the runtime. This indicates that the procedure of generating the lists is really the most influential, here. The generation phase in this case refers to the procedure of generating the lists of elements to be placed inside a particular subtree or at one leaf element of the R-Tree. This method executes the distribution of elements throughout the spatial structure.
As the amount of inserted elements in this iteration is also quite low, it is probably obvious that the effects of putting items into the secondary inverted index structures do not play a too

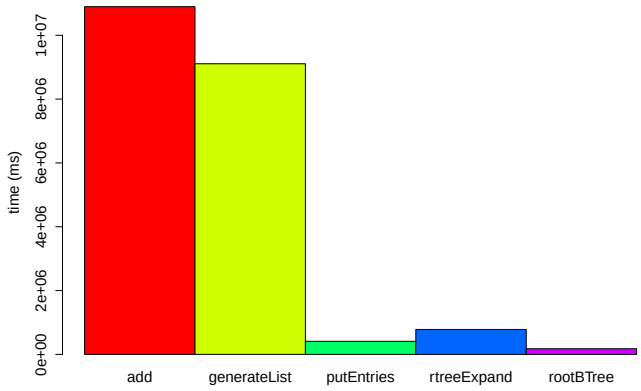


Figure 5.29: Duration of Phases

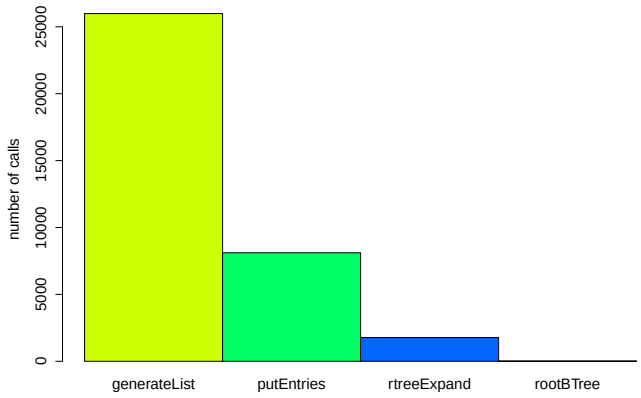


Figure 5.30: Amount of Calls

important role in this iteration. The manipulation algorithms at the initial inverted index structures and the R-Tree are not very influential either in this iteration.

The quantity of references to a particular phase can be seen in figure 5.30. This indicates that besides requiring most of the time in the entire process, the method for generating the lists is also the one called most frequently in this iteration. The other methods are by far not called quite as often as this one.

Regarding the observation that the method takes most time and is also called most often, it does presumably not consume much time per call but shows the most runtime as a method with a short duration is called very often, this also produces a large time overhead. Figure 5.31 displays the average runtime of the particular phase and demonstrates that the root B-Tree phase has the most relative runtime. However, the generation phase is far more influential with respect to the entire runtime than the root B-Tree. Hence, that this phase is the candidate for optimisation in this iteration.
The paths analyser tool is also used in this case in order to determine the exact starting points for optimisation and to analyse the effects which lead to them.

The parameter settings for iteration 1 are shown in table 5.4. The local threshold value is quite high. For this reason, the evaluation is stopped at a certain stage if two paths differ less than

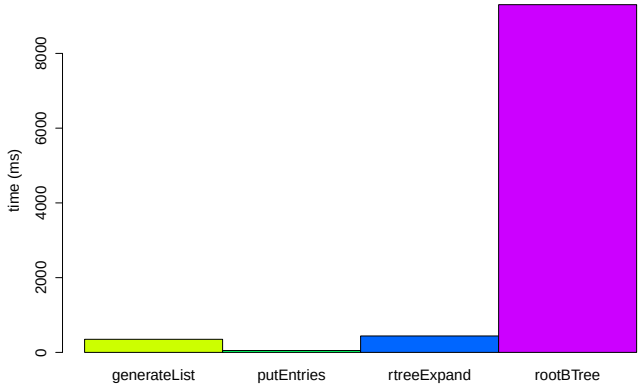


Figure 5.31: Average Runtime per Phase

Parameter	Value
Local Threshold	0.8
Global Threshold	0.04
Maximum Path Steps	2

Table 5.4: Parameter Values for Iteration 1

80% from each other as to the parental runtime. This is quite high as the method for generating the lists is substantially dominant in this iteration. The global threshold value is set to a very low value of 4%. That means that paths are kept for further evaluation if they contribute at least 4% of the parental runtime. So it is possible to keep more than one path in the evaluation as one path is really dominant here. The maximum path steps criterion is set to 2, which means that the evaluation follows the relatively most influential paths at maximum two steps until the evaluation stops and returns exactly this path.
The results from the paths analyser framework using the values given in table 5.4 are that at maximum two subpaths come into question as optimisation candidates:

- `generateList`
- `putEntries`

Due to the basic evaluation carried out with the data from the phases input, it can easily be seen that the `putEntries` method only contributes a very small amount of time to the entire process. Hence, it is obvious that the generation of the sublists for the elements is selected as a single candidate for optimisation in this iteration.
The empirical measurements only show the most influential methods. They do not specify the exact effects leading to the durations. Therefore, a more detailed analysis must be carried out based on the call graph of the total program. As it is stored inside a graph database, the values can be deduced after the actual execution of the program. Thus, a deep analysis of the basic causes for these circumstances must be carried out based on the path analyser tool.
A chart of the functions called by the `generateList` approach can be found in table 5.5. This table has a descending order concerning the total duration of the respective methods. It clearly indicates that most issues with the given approach are produced by the first four methods, `find`, `pointEquals`, `getSerializedForm` and `rangeContains`. Thus, these methods must be inspected manually in code and the effects which lead to the current situation must be analysed.
The method for loading items from the document heap is called whenever a point is loaded. That means that if one document is encountered which is referenced from a bit index to be

Function	Duration	Amount
DocumentHeap.find	5, 606, 290.209	6, 910, 709
HybridRTreePointOpclass.pointEquals	1, 457, 721.946	55, 744, 680
HybridRTreePointOpclass.getSerializedForm	605, 931.206	72, 872, 957
HybridRTreePointOpclass.rangeContains	445, 792.06	17, 128, 277
BitlistHybridIndexDocumentOpclass.getPointValues	256, 835.443	6, 913, 418

Table 5.5: Function Calls Called by `generateList`

checked at one particular R-Tree element, all respective points must be loaded in order to find out whether of the point is inside the range or the equality of two points is given in the leaf node case. Thus, a clear follow-up of this result is that even, if the find method is quite often used the methods for checking point equality or containment of a point inside a range is also used frequently. Loading one document, here, leads to loading at least one page additionally. As each document is loaded from the document heap each time it is required, this causes frequent operations on this method which is also attested by the amount of references towards this particular method. The loading of pages may take a lot of time, especially if they are not in the cache of the pages of the database system. As a lot of pages are loaded in this case, this also brings about the situation that the page cache from the H2 database must be refreshed very often and also many affected pages are not in the cache and thus must be loaded from scratch. The next effect from this method is the frequent use of functions checking the values or serialising values. In inner nodes, the method `rangeContains` is used whereas `pointEquals` is utilised in leaf nodes. As each time these functions are used, the values must be deserialized completely and this also is done frequently, the respective functions are used quite frequently. At this stage, the origin of the current performance issues is obvious. The main impact comes from the fact that individual pages must probably be loaded quite often by using the `find` method of the document heap functionality and the checks for equality or containment of the affected spatial data types. The spatial types must always be deserialized completely by the operator class in order to be checked while at each check for particular documents, they must to be loaded entirely from a database page into main memory to inspect them. Having checked the documents, they are removed from main memory and loaded from scratch for further operations.
Hence, the clear outcome of the basic observations at this iteration is the following:

1. The most influential phase requiring $\approx 84\%$ of the entire runtime is `generateList`,
2. the main problems of this phase result from having to load entire pages repeatedly in one insertion operation and
3. the checks for spatial data types contribute a lot to the entire runtime based on (de-)serialisation functionalities inside the operator class.

5.2.2.2 Solution Alternatives

Based on the data measured and analysed in the previous subsection, there are apparently issues in two parts of the reorganisation algorithm. The first issue results from loading a document to check it inside the algorithm. The documents that shall be tested are loaded each time they are queried for. The distribution algorithm checks all documents valid for the current subtree. The lists which are examined for validity and to find out whether they may be passed through the subsequent subtrees are constructed by pointing from a term id which also represents the bit index valid for the respective term to a set of documents containing the term. That means that one document is loaded repeatedly from disk if it is contained in manifold lists referred by multiple terms. Documents with many frequently used terms are likely to reside in a lot of lists, too. Therefore, they are accessed during the generation process very often as well.

This process is also executed at each element of the hybrid R-Tree for which the fitting elements must be derived. Besides this issue of having to load the documents, they are only retrieved from the disk in order to execute very basic checks. The checks are only carried out to determine whether at least one point instance stored in a particular document is contained inside the region described by one inner R-Tree element or if at least one point is equal to a certain point stored in a leaf node. Here again, for this process the checks must be executed each time for each document and element combination.
The database by itself already supports the caching of disk pages in buffers. This is done besides the actual underlying storage mechanisms provided by the file system or the mass storage devices. In modern systems using recent hardware, there are caching mechanisms on file systems and physical disk bases. These facts are hardly the subject of discussion for the performance analysis of hybrid index structures as these low level features can hardly be modified and thus are not in the focus of the software optimisation process for hybrid index structures. Thus, optimisation capabilities inside either the database (which is also not the subject of discussion as the database should not be changed according to an individual need of a storage structure) or the strategies and algorithms used inside the reorganisation process itself must be searched. Thus, solutions directly inside the algorithms of the hybrid index structure should be found, if possible. Hence, the main target for the first optimisation iteration is to detect possibilities to reduce the number of accesses to the `find` method of the document heap or to enhance the retrieval speed there.
The second issue showing up inside the analysis data is the amount of comparisons carried out inside the process. The comparisons are carried out very frequently. This refers to the methods `pointEquals` and `rangeContains` as well as `getSerializedForm`. The method `getSerializedForm` is present to serialise the data objects to a suitable representation which can then be stored inside a database page. This method is issued in order to carry out in advance the comparisons of the methods `pointEquals` and `rangeContains`. It turns out easily that the sum of references of the two individual methods (55, 744, 680 + 17, 128, 277) equals the amount of calls to this particular method (72, 872, 957). Hence, the main question about this kind of comparison is to find out if, on the one hand, the amount of comparisons can be lowered or, on the other, if the process can be refactored in order to optimise the access to the structures. At the current stage, the serialisation and deserialisation of data objects (multidimensional ranges and points) is done on each access involving a large time overhead.

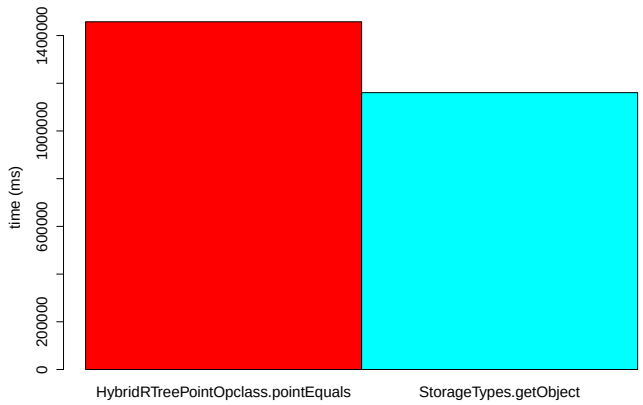


Figure 5.32: Comparison of Total Time to Relative Time of Deserialisation

Figure 5.32 shows the total time for the equals function on points with respect to the time required for deserialising it (`StorageTypes.getObject`). It is obvious that a very large amount of time relative to the total execution time of the comparison function is spent inside the deserialisation function. Similar figures hold for the ranges. For this reason, the second subject of investigation during this optimisation phase is the implementation of proper retrieval methods for the objects, too.

Summarising, it can be stated that either the amount of disk accesses for finding objects inside the document heap or the time required to do so must be worked on in this optimisation iteration. Besides this fact, also the process of the (de-)serialisation of objects for comparing the particular values must be changed in order to obtain an improvement at this stage.
The following paragraphs list the alternatives for implementing the optimizations based on the experimentally measured data.

5.2.2.2.1 Document Heap Retrieval Generally, there are always multiple options to limit the cumulative execution duration for a particular operation:

1. Reducing the runtime of one individual execution
2. Reducing the amount of executions of the entire process

The two alternatives repeatedly occur during this work and will be discussed for each optimisation step.
Reducing the runtime of one individual execution of the retrieval process seems to be infeasible in this context. The `find` operation actually is nothing but a wrapper function around loading one particular page from the database. That means that it does not do much more than just referring to the process of loading pages inside the database in order to regenerate the object lists (points) from the originally stored items. For this reason, this strongly depends on the actual runtime of the loading process of a single page. The database itself implements a caching strategy to keep the most recently used database pages in main memory for not having to load them into main memory on request. Therefore, only pages which have not been loaded recently or not at all must be loaded entirely into the database. This is a process which takes a lot of time as first the block must be found inside the program itself. Subsequently, a request for loading the buffer is sent to the filesystem. Then, the disk head must be positioned and the data are read into main memory afterwards. However, positioning and actual reading from the disk is a process which, nowadays, still takes a couple of milliseconds to be executed. If this operation must often be carried out, the required time overhead is enormous. Unfortunately, the disk buffers to be loaded are not ordered so that more than one document can definitely be found inside one page at a time. Thus, access to the respective disk buffers might be spread along a great amount of pages and, probably, very distant pages regarding the position on a rotating disk, too. Actually, optimising the runtime of one individual execution is not feasible here as it relates directly to operations which cannot be modified by the current index implementation.
The second option at this stage is to lower the amount of executions to the process. This option can be achieved by at least two methods:

1. Pre-calculation of very frequently used instances
2. Caching of accesses

The caching of accesses in this case refers to a caching operation with respect to the elements already loaded. Thus, accesses to document heap pages could hypothetically be cached for each R-Tree element, node or the entire insertion process.
The pre-calculation of instances used very frequently can also be done by utilising some kind of caching mechanism. This could involve the pre-calculation of documents which are loaded in many cases. The time frame in this case again is at element, node or insertion process level.
The decision made for this step is the setup of a hybrid strategy which tries to cache accesses to document heap pages using a pre-calculation of the items to be loaded.
Caching is a widely used technique for disk access. It is implemented directly inside the filesystem (see, e.g. [96, page 396 ff.]). Besides the filesystem, there are also caching mechanisms for database systems set up on top of filesystems. Already in early developments, database systems should be able to cache accesses to fixed size disk buffers (see, e.g. [27, 10]).

Nowadays, popular strategies for caching data of disk buffers include the LRU-K cache [80] or the two queue strategy [55] arranged on top of the LRU-K cache. Extensible strategies to adopt general purpose caching strategies to customised problem statements or access structures are also defined [14]. These strategies allow customised access structures to modify the caching strategy in order to adopt the strategy to the need of the access structure. That means that basic algorithms may be provided with hints in order to function in a better way with customised index structures.
The LRU-K cache is a structure applying the least recently used caching metrics. For this reason, only the most frequently employed data objects are kept inside the data structure whereas the least frequently used ones are acceptable for deletion. This strategy does not prove to be optimal as statistical approaches perform better in most cases. There are optimal caching strategies which show optimal performance. However, a lot of knowledge must be included in the setup which has to do with the expected next page accesses. Yet, most of the time this knowledge is not present to the cache to adopt it directly to the requirements of the program. The least recently used family of caching strategies has proven to be sufficiently successful in the last years. This strategy is already employed in a lot of database related applications. The H2 database, which serves as a basis for the currently inspected analysis, already applies this strategy internally for determining the pages to be removed. LRU works especially well when referring to size limited page replacement strategies. Hence, a fixed size buffer is used which is managed according to the LRU strategy. Thus, it is truncated with respect to a certain predefined metric if a certain predefined limit of buffers is reached. There are strategies to remove only one item. However, in all likelihood, for each item a new insert and remove operation is issued if a lot of new objects, currently not residing in the cache, are queried for. Another strategy is to truncate the cache to a predefined minimum amount of elements from the back of this cache.
In general, caching seems to be a reasonable approach in this context. Besides caching, also a pre-calculation strategy comes into consideration. That means that the assignments of documents pointing to individual data objects (points) could also be pre-calculated inside, e.g., an associative array. There is also a possibility of extracting the entire point list first and then distribute the respective points to the particular documents they occur in. This strategy limits the amount of main memory in comparison with instantiating each data object for each document employing duplicate points in main memory. It must always be kept in mind to limit the amount of main memory required by the storage types to reduce memory consumption to a reasonable low limit. Thus, size limited caches are a meaningful approach for importing data in main memory and remove the objects with a low likelihood of being accessed thereafter. These are the two alternatives to be chosen for implementation at this stage. The concrete selected solution is discussed in subsection 5.2.2.3.

5.2.2.2.2 (De-)Serialisation Optimisation At the current stage of the optimisation iteration, the (de-)serialisation of data objects under test is one of the most influential subalgorithms, besides the lookup of respective objects. The first iteration stores all items in serialised data portions which must be deserialized on each access and serialised after change in order to fit into a particular data page. It stores composed objects which are sometimes quite complex. The hybrid R-Tree elements, e.g., are composed of a spatial object, the bitlist and a reference to a secondary inverted index or a child page. Therefore, before comparing a stored object with a new one to be inserted, the object must first be deserialized completely from a database page and then compared. After this comparison (either point inside range or point equality), the results from the (de-)serialisation process are removed from main memory, anew.
This is basically the stage where the optimisation is likely to succeed. Instead of removing the (de-)serialised version from main memory, again, the data object could be stored in some kind of specialised storage structure temporarily. Hence, additional accesses may first query the specialised storage structure for presence of the particular item and subsequently retrieve it from there or create a new object storing it to this structure afterwards if it is not present yet. This operation may, again, be solved by the use of caches. Another option is to change the storage structures so that they are faster to (de-)serialise. Yet, at the current stage, first the caching mechanism is tried out.

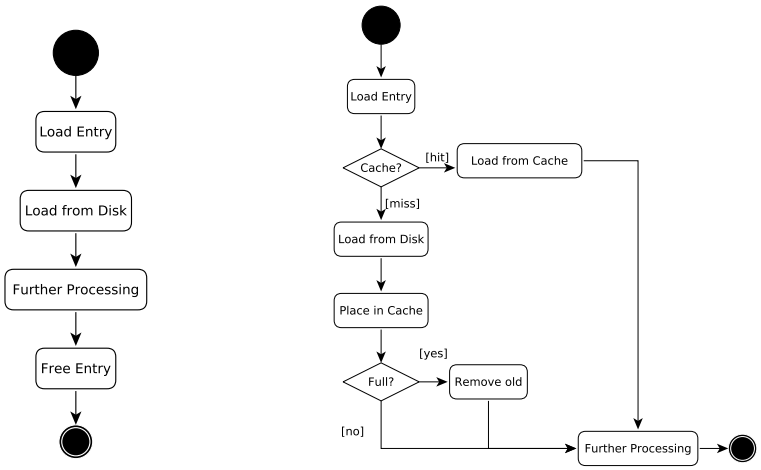
There are substantial differences to disk page caching when regarding this situation. Disk page caching relies on a fixed size of the respective storage structures. Each disk page requires a predefined amount of memory (in disk and main memory). If, e.g., a 2 KB page size is given, the amount of memory on disk is 2 KB and the amount of memory in main memory is a constant of similar size, probably of a little more than 2 KB based on management overhead. However, it may be assumed that these structures are still calculable and of nearly fixed size. In memory storage, especially in managed languages like Java, this is different. The individual components stored there must be determined internally to sum up the space requirements of data objects. Therefore, it is necessary to execute pre-calculations for all items to ascertain the actual data size. Especially the space requirements of lists must be determined in order to be able to give a space estimation here. The main target is not to limit the amount of items stored but the size of the items. Hence, a pre-processing step must be carried out which calculates the size of the respective data items.
A LRU cache strategy is an option here, as well. However, not only the amount of items currently stored inside the cache structure but also the size of the respective storage items must be kept in mind. Hence, the item(s) are deleted from the end, as given in the general LRU strategy, and counters must be maintained for the amount of items as well as the required main memory storage amount.

These are the strategies which may be selected for improving the performance of the hybrid index structure at this stage. The concrete selected solutions and their implementation are described in the following subsection.

5.2.2.3 Selected Solutions

This subsection describes the actually applied solutions and the causes bringing about the decisions to implement them. The concrete implementation details are also outlined here, together with the effects expected as outcome of the changes.

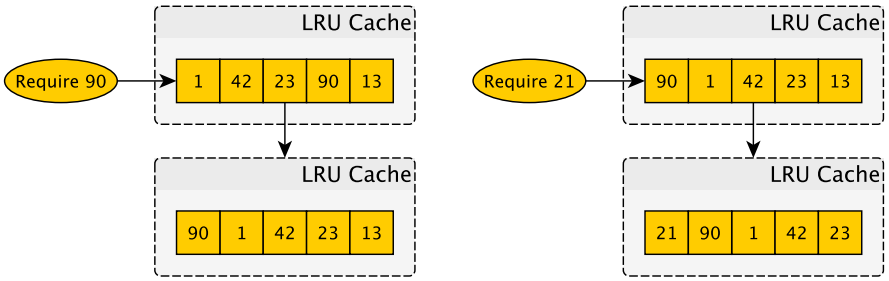
5.2.2.3.1 Document Heap Retrieval Regarding the document heap retrieval, there is no possibility of enhancing single accesses to the respective method. For this reason, the decision is taken to wrap the accesses in order to lower the amount of actually loaded nodes. That indicates that a caching mechanism is set up which will limit the amount of calls to this particular function. Besides the caching, also an adapted pre-calculation of items is carried out. Two basic caches are introduced in order to enhance access to the respective functions. One cache is used for storing the point values individually. The first storage structure is applied to collect all affected points for the current reorganisation run. This structure consequently represents a set of all points contained in the set of documents to be processed in the current run. It is present in order to avoid duplicates. That means that each affected point is not held in memory multiple times but exactly once, even if it occurs in multiple documents. This action is taken in order to lower the amount of memory required for one insertion. It is important to say that all point values are stored in this structure permanently (for the currently inspected run). Therefore, it does not represent a real caching strategy as items are not removed from there in case of certain events. Thus, this structure is not limited to a certain amount of memory used at maximum. Assigning points to documents from this cache in order to avoid repeated instantiations simply performs a lookup on this cache and attributes the respective point to the document.
The second action taken at the document heap is the introduction of an actual cache structure. There are studies (e.g. [74]) demonstrating that the LRU strategy is shows an acceptable performance in average case. Furthermore, it is also implemented very frequently and set up, e.g., in modern database and filesystems. Based on the wide spread use and the acceptable performance, there are even cases where, e.g., least frequently used (LFU) or a modified LFU algorithm called "File Length Algorithm (LEN)" outperform this strategy, this cache type is



(a) Without Cache (b) With Cache

Figure 5.33: Process of Loading Elements from Disk / Buffer

implemented, here. However, the differences due to the empirical measurements given in [74] are not too big. Thus, the decision at this point is to implement a LRU strategy.
Figure 5.33 shows the process of loading elements from disk using deserialisation and processing them. Subfigure 5.33a shows direct loading and processing of items. In this process, the data are loaded, processed and subsequently freed again. Subfigure 5.33b, however, includes cache processing. Two paths may be visited, there. If the requested item is already in the cache, it is directly taken from there and processed further. In case the item is not inside the cache (cache miss), it is loaded from disk and placed inside the cache. Provided the cache is full, old items are removed before placing new ones there and the processing is continued. This procedure includes the benefit that the cache may contain size restrictions and if the check of the cache being full evaluates to true, old items may be removed. The metrics of the full check may vary depending on the actual cache implementation.



(a) Hit (b) Miss

Figure 5.34: LRU Cache Strategy

The concrete implementation of least recently used caching utilises a queue to represent the order of entries. On a cache hit, the item found inside the cache is placed at the head of the queue. This procedure can be seen in subfigure 5.34a. The other case is the cache miss. That means that the item to be found is not currently stored inside the queue representing the cache content. Then the "missed" item is stored at the head of the queue. If the storage restrictions of the queue exceed the limit (either memory or amount) old items are removed from the tail. How many items are selected for removal depends on the deletion strategy for the cache implementation. Subfigure 5.34b shows the situation of a cache miss as well as an overflow. When the item 21 shall be placed inside the cache, the cache with a capacity of 5 becomes

overfull and the eldest entry is removed (13) whilst the new item is placed at the head of the cache queue.
Figures 5.33 and 5.34 show the general process of loading items with and without cache. It needs to be adopted to the actual requirements of the software. The cache given in the illustrations only stores individual numbers and is implemented as a mapping function which points from a certain key to an individual value. In case of document heap requests, the structure stored is a mapping from the row key representing the respective document heap item individually to a list of points stored there. Thus, the cache request lookups query for the row key. If present, the list of points valid for the document are returned to the query issuer. The cache implemented here is able to limit the amount of memory spent during this procedure. Therefore, it is possible to narrow the memory limit down to a predefined threshold which may be specified during the instantiation of the LRU cache. This cache implementation is unidirectional, which means that the request only may call for the row key returning the list of points contained. The respective caches are cleared after each insertion operation.
The implementation variants chosen for document heap retrieval optimisation are the following:

1. Storage of individual points from all affected documents in an associative array in order to avoid multiple instantiation for individual points occurring in numerous documents for intermediate main memory reduction and pre-caching of all points
2. Implementation of LRU caching in order to keep the least recently used documents in a cache which may be limited by a maximum of main memory to be allocated there.

5.2.2.3.2 (De-)Serialisation Optimisation The (de-)serialisation optimisation approach is addressed in a similar way as the document heap retrieval. In this case, similar facts as for document heap retrieval are valid. The procedure of serialising or deserialising particular items may not be improved as external methods are used for this task. Therefore, the amount of (de-)serialisation operations must be lowered. However, at this step, it is not possible to reduce the number of calls to the comparison functions (`rangeContains` and `pointEquals`). So, the only possibility of optimising this is to intercept the calls between the (de-)serialisation and the actual comparison. The amount of comparisons cannot be changed as the distribution depends on the actual distribution function and therefore the comparisons must be executed at these exact phases inside the algorithm. That means that somewhere in between the call to the respective comparison function and the actual comparison which is executed after deserialisation, the action must be included to optimise the (de-)serialisation process.
The solution at this stage is to include another caching mechanism. The cache included here is a bidirectional cache as items are retrieved either from the deserialized to serialised version or vice versa. Therefore, in order to be able to support both kinds of accesses, the cache used here is implemented in a bidirectional way. Thus, the cache operation contained there can be seen as a proxy interceptor between the call to the comparison, the deserialisation and the actual execution of the comparison. This bidirectional cache is implemented for each storage structure affected inside the hybrid index structure. That is to say that caching is available for document heap, initial inverted index, hybrid R-Tree as well as the secondary inverted index structures. So, all of the storage structures affected will, most probably, benefit from this change as every kind of structure can be cached then.
The main work, besides using the LRU cache approach, which is already done in the document heap retrieval implementation, is the derivation of the required main memory space for each storage structure. Thus, a structure is included which has the possibility to derive the main memory requirements needed for each type or object to be stored. There is a mechanism to derive the storage of predefined types which may be registered to a size derivation procedure. This is especially important if structures are inspected contributing dynamically growing elements like arrays or lists as the number of individual elements stored inside these structures is not known.
Summarising, the selected solution relies on a bidirectional storage structure cache present to cache the (de-)serialisation executions to limit their amount. A freely configurable amount of main memory may be set in this case in order to avoid memory overruns in the implementation.

5.2.2.4 Evaluation

The evaluation to be carried out on the optimisation iteration(s) includes a post-test which is executed with exactly the same data as the initial pre-test for the respective iteration. Therefore, the outer circumstances of the pre- and post-tests must be exactly the same. The circumstances may, however, change if they affect the performance and are subject of investigation for the solutions. That means that at least the amount of documents stays alike. As other outer circumstances do not change in this iteration, everything stays unvaried compared to the pre-tests. Hence, the post-test analysis is carried out on data measured with the new versions of the hybrid index structure. Referring to the model approach, nothing changes in the model for the first optimisation iteration. In comparison to the initial tests, first, a general analysis is realised regarding the entire reorganisation procedure. Subsequently, the respectively optimised individual parts of the algorithms are evaluated individually.

5.2.2.4.1 General Evaluation The effects of the optimisation on a total overview basis is displayed in figure 5.35. It describes the effects on the entire reorganisation process.

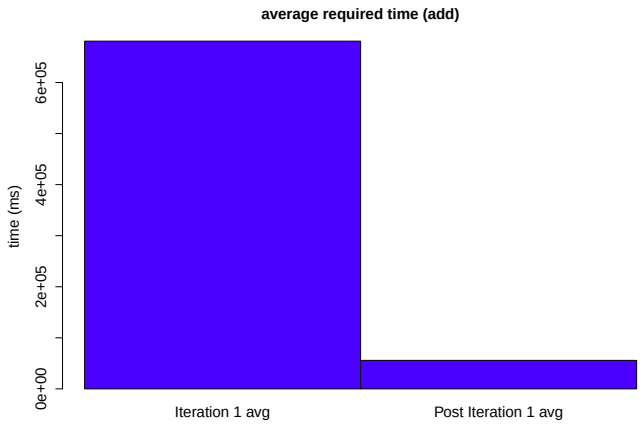


Figure 5.35: Add Function Comparison

Therefore, it displays the duration effects on the function `add` which is responsible for inserting an item to the index after it has been added to the actual data table. It can be seen that the effects are enormous in this iteration.

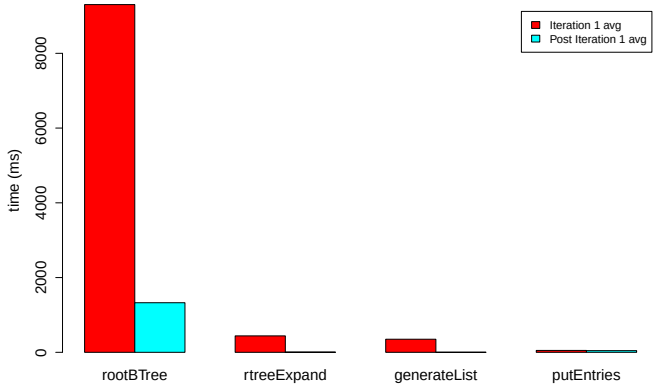


Figure 5.36: Total Overview of Iteration 1 Comparison

A total overview of the measurement data from iteration 1 can be seen in figure 5.36. Each of the respective particular phases is inspected in this evaluation. Actually, the putting phase is not affected so much in this case. This is based on the fact that at this stage of the index structure, the amount of elements added to the secondary inverted index in the putEntries phase is not so big. Therefore, the effects are not as visible as in the other phases here. The main targets to be optimised now are the R-Tree phases `rtreeExpand` and `generateList`. From this overview graphics, it can be seen that each phase gains improvements as a side effect from the optimisation iteration.

5.2.2.4.2 Individual Evaluation The individual evaluation is carried out in order to validate the individual effects on the respectively optimised phases. Thus, the two phases of `generateList` and `rtreeExpand` are of special interest here as both of them directly refer to the cache.

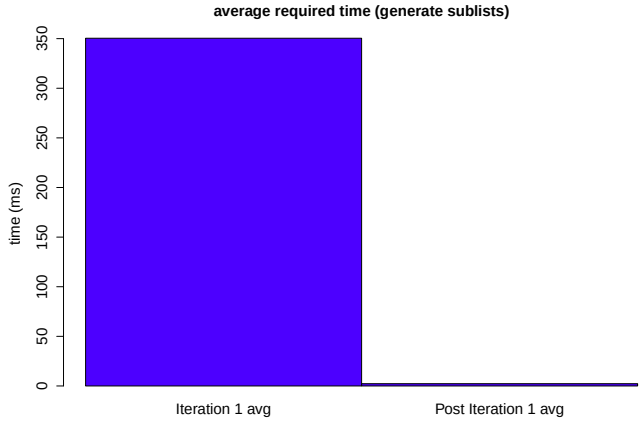


Figure 5.37: Generate Function Comparison

The comparison graph of pre (left) and post (right) can be seen in figure 5.37. It shows a very obvious advantage of the improved version of generating the list in contrast to the previous measurements.

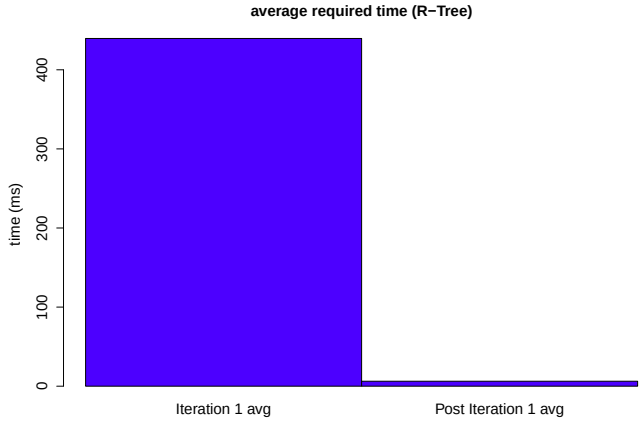


Figure 5.38: R-Tree Modification Function Comparison

The R-Tree expansion function which distributes the new points to the R-Tree (figure 5.38) also reveals enormous improvements compared to the initial version. It can easily be seen that the effects described in the previous subsection improve the runtime a lot.

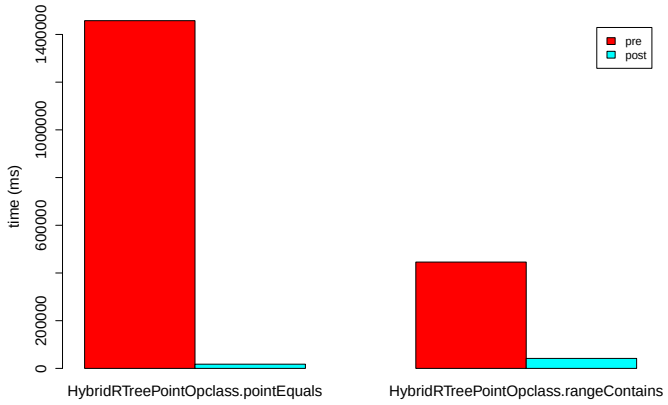


Figure 5.39: Duration of Comparison Functions

However, the detailed comparison of the runtime must still be carried out here.
The comparison of the respective functions for equality of points and containment of points inside ranges can be seen in 5.39. These figures are basically the results of the caching implementation.

Function	Duration Before	Duration After
HybridRTreePointOpclass.pointEquals	1,457,721.9	17,678.65
HybridRTreePointOpclass.rangeContains	445,792.1	41,841.89

Table 5.6: Comparison of Pre- and Post-Tests

Table 5.6 summarises the numerical representations of the given function calls which are depicted in figure 5.39. The function `pointEquals` only takes $\approx 1.2\%$ of the runtime as compared before, whereas `rangeContains` takes $\approx 9.4\%$ of the total runtime in comparison with the previous execution.
The load operation of elements from the document heap entirely changes in the new approach. The potentially size limited cache which is used without any restrictions replaces this operation. Therefore, the point values stored inside this cache are obtained by a load operation executed once per reorganisation operation. Hence, the respective function is only called very seldom in this approach.

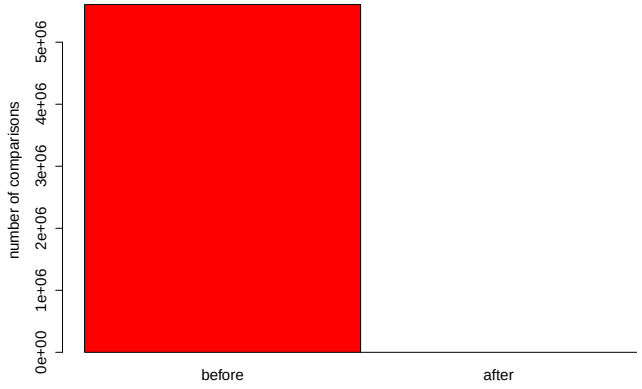


Figure 5.40: Duration of Finding Items in Document Heap (total)

The cumulative duration of the respective function calls for finding items inside the document heap can be seen in figure 5.40. It is obvious that the effect of this optimisation is tremendous. The exact values for the given functions are demonstrated in chart 5.7. The table shows that the

Function	Duration	Amount
DocumentHeap.find before	5,606,290.209	6,910,709
DocumentHeap.find after	22.524	2,183

Table 5.7: Exact Times and Amount of References for Finding Items inside the Document Heap

amount of references to the respective function lowers dramatically as does the required time to find the particular items. The reason for this is that the documents are kept inside a cache structure during the reorganisation once loaded from disk. They are only present in this cache for the time frame of one reorganisation run and are removed afterwards. Therefore, this structure is only used as temporary memory while one document is inserted.

5.2.2.5 Summary

The first optimisation iteration basically finished by introducing cache functionalities at different stages of the reorganisation procedure. This indicates that the internal structure of the hybrid index did not change at all. Only the algorithms but not the storage areas altered. Thus, regarding the retrieval operations executed on the hybrid index, no modifications are necessary in order to query for the respective items again.
The retrieval of elements must always be kept in mind, because changes in the internal storage mechanisms of the index structure could, hypothetically, also modify the retrieval behaviour and performance. Nevertheless, as only caching and a translocation of certain storage areas into the main memory have occurred, the query behaviour stays untouched. However, based on the cache introduced for item (de-)serialisation, there is also the possibility of increasing the query performance, at least regarding time, owing to the changes in the reorganisation procedure. The amount of loaded nodes stays the same for the respective operations, because only the internal algorithmic has been changed which has not affected the disk pages required during the reorganisation runs, except the document heap pages during insertions, obviously.

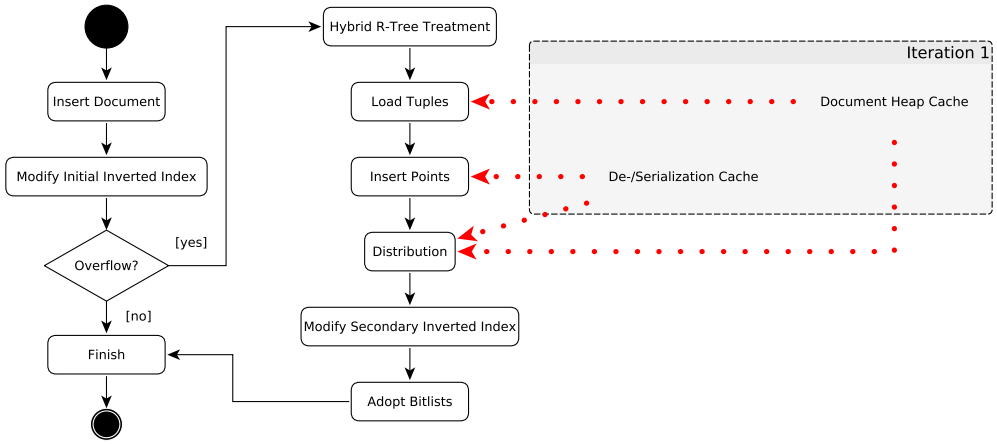


Figure 5.41: Activity Diagram of Changes Performed in Iteration 1

A summary of the algorithmic changes executed in iteration 1 can be depicted from figure 5.41. It mainly shows the introduction of the respective LRU caches and the phases they apply to. The document heap cache is utilised each time a tuple is loaded from there. That means that it

is used at the R-Tree expansion phase where the tuples are loaded to place each of the respectively affected point values. In addition, the distribution algorithm takes the document heap tuples into account when checks are executed to find out whether one specific tuple is valid for one R-Tree element.
The second cache included in this iteration is the (de-)serialisation cache. It applies every time a tuple needs to be loaded from a page. It mainly applies to the R-Tree modification phase but is also partially used in other phases like the inverted index manipulation. Yet, it is focused towards the R-Tree manipulation and also shows the greatest effects, there.

5.2.3 R-Tree Distribution Considerations

The R-Tree [36] is a well-known storage structure for saving spatial data. It can cope with extended data like geographical regions by deriving rectangles or local point data to be stored. In the hybrid index, the R-Tree is used for the storage and retrieval of the multidimensional point values as well as the combinations with the bitlists which care about the indication of presence or absence of a certain term in a predefined subtree. The R-Tree basic algorithm optimises the node regions regarding the area covered and distributes elements in order to minimise the area occupied by the respective node. Yet, it has been noticed that optimising the nodes with respect to the area covered by them is not the best way to achieve a high performing retrieval complexity. Several other approaches have been proposed to improve the distribution of items to reach a better performance than the baseline algorithms. They comprise the R*-Tree [5], the R+-Tree [92], the revised R*-Tree [6] and a new linear [3] splitting approach. As the R-Tree is one of the most important structures in the application of the hybrid approach, the distribution of the items must be checked to improve the retrieval throughput. Based on the fact that the distribution algorithm of items to proper elements of the R-Tree also performs search like operations, an improvement of the reorganisation performance is expected here, as well.

The second iteration is executed due to outcome of the first iteration. The basic setup of the test suite is exactly the same as in the first iteration (pre- and post-tests). However, based on the optimizations of the first iteration, the amount of elements inserted to the hybrid index has been increased dramatically. The amount of measurements is increased to 39. The total amount of executions of the test suite is therefore incremented to 78. Hence, on grounds of equation 5.5, the amount of items inserted sequentially is $n = \lceil \frac{78}{2} \rceil \cdot 50 + \lfloor \frac{78}{2} \rfloor = 1989$. The execution of the test suite repetitions is terminated manually because of time constraints.

Parameter	Value	Explanation
HLimit	200	Artificial Upper Bound
Element Count	5	Amount of Elements per R-Tree node
Document Count	1989	Amount of Documents to be inserted
Dataset	Wikipedia	Dataset to be processed (Wikipedia sorted by ids)
Split	R-Tree Split	R-Tree splitting method
Choose Subtree	R-Tree Choose Subtree	Method for Selecting a proper subtree to place an element in

Table 5.8: Settings for the Pre-Tests Test Suite Run

The detailed parameter settings can be seen in table 5.8. The main distinction to the one used in iteration 1 is the changed document count parameter. Based on the outcome of the first optimisation iteration, the amount of documents processed for this iteration could be doubled. Yet, the test suite is again killed on account of the time constraint required to execute the entire run.
In contrast to the first iteration which started with a set of pre-tests and continued with a post-test which was evaluated regarding the initially taken data from the pre-test, henceforth each iteration is analysed based on a pre- and post-test of the particular iteration. The parameter settings must by no means be changed between pre- and post-test, except that (at least) one of these is subject of change during this optimisation iteration.

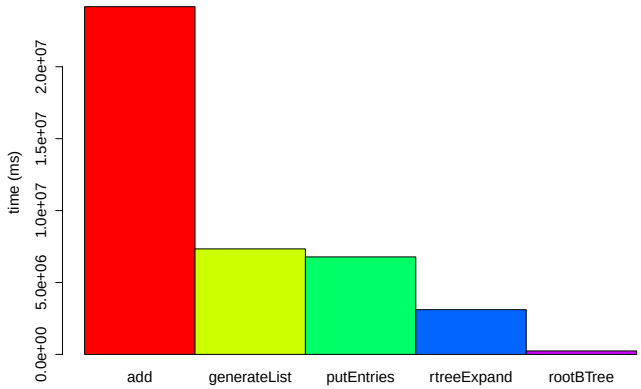


Figure 5.42: Duration of Phases

5.2.3.1 Basic Observations

The basic observations for this iteration start with an analysis of the respective phases used in the reorganisation process. The phases are the same as in iteration 1.
Figure 5.42 shows the duration of the particular phases during the reorganisation of the hybrid index structure. In this iteration, the phases of putting the entries to the secondary inverted index and generating the lists of elements to be distributed to a subtree or put to a secondary inverted index structure are the dominating ones regarding the total duration. They require nearly the same amount of time. Expanding the R-tree takes about half of the total time necessary for generating the lists. The modification of the initial inverted index (`rootBtree`) does not have much influence on the total runtime. Therefore, generating the lists, putting entries to secondary inverted index structures and preparing storage structures inside the R-Tree are candidates in this case.

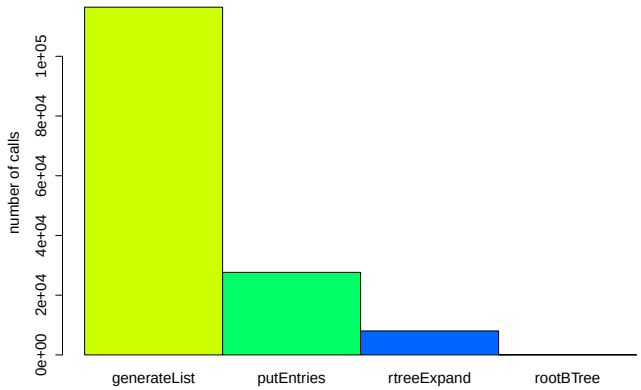


Figure 5.43: Amount of Calls

The amount of executions of the respective phase can be seen in figure 5.43. It displays that obviously most calls are targeted towards generating the list. The root B-Tree phase is not of particular interest in this case as it is not called very often. The second place in this analysis is to put entries to their final destination. Yet, the percentual amount of calls to this phase is only $\approx 23.73\%$ as to generating the lists. The R-Tree expansion phase is executed $\approx 29.04\%$ regarding calls to put the entries.

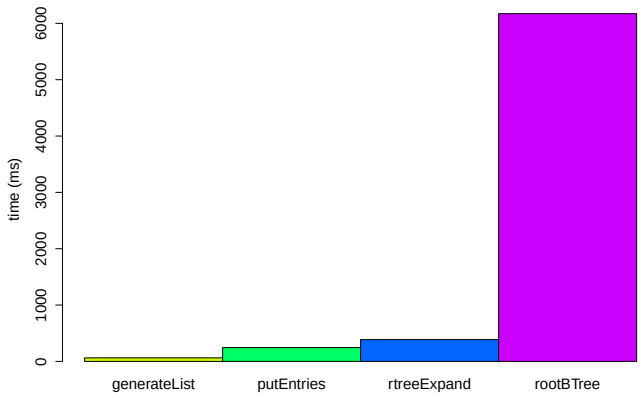


Figure 5.44: Average Runtime per Phase

Average runtime values for each phase can be seen in figure 5.44. It displays the longest one at the initial inverted index manipulation phase. However, regarding the total runtime as well as the amount of calls to this function (see figures 5.42 and 5.43), it does not have much influence on the entire reorganisation process. Therefore, the remaining phases stay of particular interest as they are basically integrated closely. The R-Tree can be expanded in a lot of different ways. There are alternatives to distribute the items throughout the tree. The distribution of the items inside the tree strongly relates to the phase of generating lists of elements fitting for particular subtrees. Finally, every item which is passed to a leaf node is placed in the respective secondary inverted index by using the `putEntries` phase. Thus, these three functions are closely connected. The expansion phase has the longest average runtime with respect to the three phases.
As a final result from this preliminary analysis of the phases it can be stated that the list generation, putting and expanding of the R-Tree are candidates for optimisation as at least generating the list and putting the entries require the majority of the entire runtime and the list generation and expansion (distribution of items) are closely related to each other.
In addition to the basic phase analysis, the paths analyser tool is utilised here in order to determine potential candidates for optimisation based on the path pruning approach.

Parameter	Value
Local Threshold	0.1
Global Threshold	0.2
Maximum Path Steps	3

Table 5.9: Parameter Values for Iteration 2

The parameters for the analysis run of iteration 2 using the paths analyser tool are outlined in table 5.9. The local threshold in this case is quite small, which means that the execution is stopped if the duration of two child function calls relative to the parental function call only differs by 10%. The global threshold does not prune too much as it leaves all calls in the evaluation which require at least 20% of the parental runtime. The maximum path steps criterion is set to 3 in this case, which means that three steps may be taken on the path with the longest duration here, until the evaluation stops.
The evaluation applying these parameter settings shows that the following parts are the most influential in this phase:

- `generateList`
- `putEntries`

In this case, the situation is comparable to the one found in iteration 1. However, as already seen in the general analysis, the differences between the two phases is not as essential as before. Hence, the two methods do not differ much.

Function	Duration	Amount
generateList	7,335,648.877	116,455
putEntries	6,778,586.915	27,639

Table 5.10: Function Values for Iteration 2

The detailed values for the function call duration and the amount of executions of the respective function can be seen in table 5.10. These two values only differ by $\approx 7.6\%$. Hence, they are very close to each other. It is obvious that the local threshold criterion set to a difference of 10% regarding the parental runtime is the one which triggered the analysis to stop here. At this stage of the project, it is not precisely defined how to optimise the performance of inserting the respective entries to the secondary inverted index. Therefore, the decision is taken to look more closely into the method of generating the lists valid for the respective subelement.
At this particular time, the second option is neglected and will be looked at in further optimisation iterations. Thus, the focus for this phase is still the optimisation of the list generation as this operation is not fast enough, yet.

5.2.3.2 Solution Alternatives

The analysis has provided two different results of optimisation candidates. This turns out with the phases analysis as well as the tool driven analysis. The selected phase or function to be optimised in this stage is the list generation as it is still not efficient enough. Hence, two ways are possible now. One option is to abbreviate the total time required for the analysis and another is to reduce the amount of references to one specific call. Reducing the total time at the current stage is no option as the previous optimisation iteration had exactly this target. Thus, the second option, i.e. reducing the amount of executions of the particularly inspected method must be taken into account.
The basic situation here is that the particular method is utilised for generating lists of entries which shall be placed inside the given subtree pointed to by an element. This element is an R-Tree element representing a multidimensional range in the inner node case or a multidimensional point in the leaf node case. The amount of items which are checked against one particular element is introduced based on the overflow treatment of high frequently occurring terms inside the initial inverted index. Therefore, this cannot be changed in order to optimise here. The quantity of checks can only be manipulated by reducing the number of potentially valid elements inside the R-Tree part. It is probably worthwhile investigating the R-Tree a little bit more in detail here.
The R-Tree is used in the initial implementation applying exactly the same metrics as proposed in the initial paper [36]. There are two crucial functions which can be employed to optimise the node structures of the R-Tree. When a new item is inserted into the R-Tree, the way how to find a proper leaf node to place the item in may be modified because of certain heuristics. Besides this operation, it is also possible to change the method of distributing the elements of one node to two new nodes if one node becomes overfull and thus must be split in two. The original algorithm proposes a distribution based on the coverage area of the nodes. Consequently, the optimisation criterion for choosing a subtree and splitting a tree node is to build dense clusters. That means that new entries are included in a subtree if they comprise the least area enlargement covered by the respectively inspected node.
However, it has shown that this criterion does not fit in all cases. The main problem with this heuristic is that only the area of a node is looked over. Other criteria like the node shape or the amount or area of overlaps are not examined. One main criterion investigated by newer approaches is the overlap. It is a well-known property of R-Trees that overlaps contribute negatively to the search time complexity. This is mainly due to the fact that in case of searches if

the search region overlaps a particular rectangular element in inner nodes, this node is candidate for results in the respective subtree. If there are overlaps between node elements, multiple nodes must be searched for results for solely one search region. Hence, the search complexity rises if overlaps exist.
There are a lot of different methods. The most widely used variations for splitting nodes and choosing a proper subtree are implemented for the purpose of evaluation.
The most common used methods for choosing a subtree are:

- R-Tree linear method [36], explained in 2.1.3.1
- R*-Tree [5], explained in 2.1.3.2
- R*-Tree revised [6]

The respective methods for splitting nodes are the following:

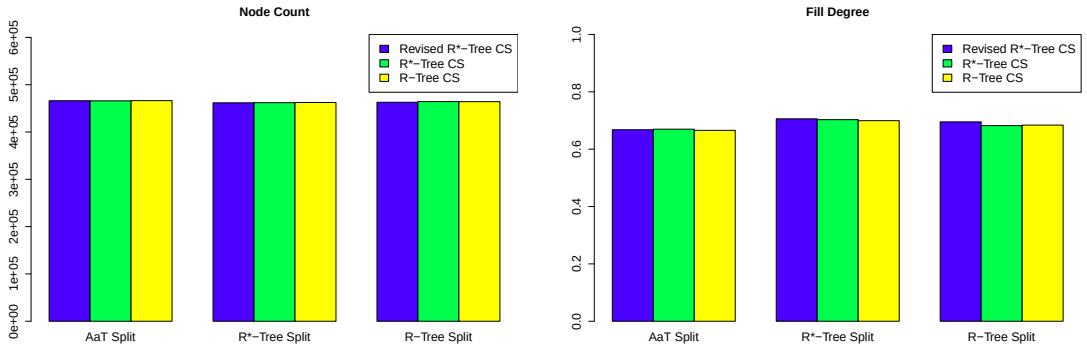
- R-Tree linear method [36]
- R*-Tree [5]
- Split Method of Ang and Tan [3], see section 2.1.3.3

The basic methods for selecting a proper subtree to distribute new items to or to split multiple nodes are explained in section 2.1.3.
One new method for choosing a proper subtree is the revised version of the R*-Tree. The implementation of the respective split method is omitted as it involves additional storage structures. Additional storage structures also require additional space inside an R-Tree element. Inside the hybrid R-Tree elements, besides the actual ranges describing the geographical features, there are the bitlists. As they include additional storage elements they need more space which may not be supplied because the bitlists would must be shortened subsequently. Since the bitlists represent terms using a bit id, less terms could be stored in one R-Tree instance if the additional identifiers required by the revised R*-Tree metrics were included.
In the leaf level, the original R*-Tree method only inspected the overlap enlargement which would be introduced when placing an additional element inside the currently inspected node. The new version optimises the overlap enlargement in inner nodes. If no overlap free distribution can be found, other optimisation criteria (volume and perimeter enlargement) are taken into account to bring about a final decision.
The given distribution methods need to be evaluated against each other in order to create storage structures using reduced overlap. This is also a crucial task inside the hybrid index. The efficiency of the insertion algorithm can be increased if overlap free distributions of items can be found. The main cause here is the list generation during the generation of lists of entries valid for a given R-Tree element. Basically, the distribution algorithm iterates recursively through the R-Tree and assigns lists of entries available for an element. This operation is comparable to searches inside the R-Tree. Therefore, if distributions with a very low amount of overlapping nodes exist, the lists tend to become more precise, too and spatial regions which probably contain a certain point value get more distinct, as well. For this reason, the distribution algorithm does not need to revisit spatial regions.
Based on this argumentation, an empirical study is required in order to determine the combination of split and choose subtree approaches which serves best for a given set of input values.

5.2.3.2.1 Empirical Evaluation of R-Tree Variants The empirical evaluation of R-Tree variants is carried out on three different datasets. All combinations of potential candidates must be checked. Regarding three variations of choosing a proper subtree and three of splitting a node makes a total of nine configurations under test. Therefore, a test suite is developed which executes the tests in order to find the best combination. The main parameter under test for R-Tree optimisation, here, is the overlap reduction. However, two separate aspects of reducing the overlap value must be inspected individually. First, the amount of overlaps needs to be evaluated. If it is very low, there is the tendency that very few parts of areas must be examined multiple times while descending into the tree. The other aspect is the total size of the overlap area. Even if only few overlaps arise, they are still likely to span over the entire area covered by the tree. Hence, it should not be assumed that reducing the amount of overlaps already solves the problem. The area of the resulting overlaps must be inspected in depth, as well.
Therefore, the following parameters are measured in this empirical study:

1. Amount of Overlapping Node Elements
2. Area Covered by Overlaps

Besides these two main criteria, also additional ones are checked. On the one hand, the total amount of nodes is measured and on the other, the fill degree is measured. The fill degree parameter is examined for a given maximum capacity of elements. That means that this parameter evaluates the amount of elements with respect to the total capacity of one particular node. If a higher fill degree can be achieved, trees tend to have a larger branching factor. This feature is desirable as it lowers the amount of levels inside the tree and thus less nodes must be loaded while searching. The amount of nodes is directly connected to the fill factor criterion. If the fill factor lowers, the amount of nodes will rise and vice versa.
The datasets under test are basically related to the input data for the optimisation iteration test runs (see section 5.1.3). For the assignment of the points to the respective articles, a gazetteer is utilised. This gazetteer is taken from GeoNames[10]. The articles get assigned specific portions from that dump. That is true for both datasets, Wikipedia and Reuters. Therefore, the articles comprise different excerpts based on textual occurrences of toponyms. As test data input for the respective test runs, the individual point data from Wikipedia and Reuters articles as well as the entire Gazetteer data are taken. It must be noted that the Gazetteer data are pre-filtered due to requirements from a previous project. In this project, duplicate toponyms as well as places having certain properties like the minimum amount of inhabitants were filtered out from the entire list.
Based on this assignment, the Reuters dataset contains 10,303 point values and the Wikipedia dataset contains all 396,300 from the original Gazetteer. Obviously, the Gazetteer dataset provides all point values. Although the Gazetteer dataset comprises the same values as Wikipedia, the datasets are not the same. Regarding the R-Tree (or variants) and distribution of elements inside the datasets, the order of the insertion of elements also plays an important role. The order of elements of Wikipedia and Gazetteer are totally different. The Gazetteer is ordered based on the original import from the dump whereas the Wikipedia point dataset is sorted due to its initial derivation from the Wikipedia dataset. The Wikipedia dataset contains all articles sorted by their id and thus the dataset is ordered based on the first occurrence of the respective toponym which gets a certain point value assigned because of their occurrence in the list of elements which results from sorting by id. Therefore, though sharing a common data base, these three datasets are basically very different.
The tests are executed in a main memory simulation where each of the approaches is implemented. Measurements for the two primary and two secondary parameters of interest are only taken once at the very end of the insertion procedure to derive the final values for each of the parameters. This procedure is carried out for each of the nine configurations and the values are exported to be analysed.

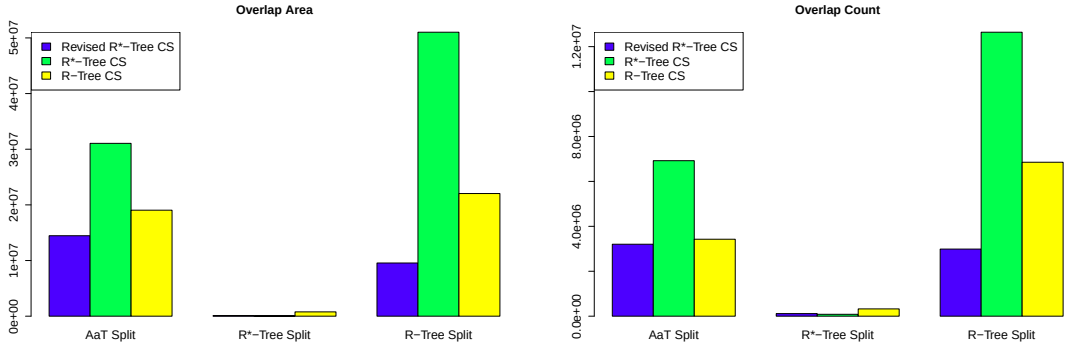


(a) Gazetteer Node Count (lower is better)

(b) Gazetteer Fill Degree (higher is better)

Figure 5.45: Gazetteer Node Count and Fill Degree

5.2.3.2.2 Results The results from the given empirical study are presented in this section. First, the data from the Gazetteer data are outlined.
The fill degree and node count for the Gazetteer dataset is demonstrated in figure 5.45. The differences of these parameters are merely marginal, here. However, it can be seen that either R*-Tree or the revised version comes off as the winner in this figure. The particular figures show bar plots for the split algorithms separated by the respective choose subtree approaches. The basic R-Tree method looses in each inspected measurement, at least regarding the choose method. Concerning the fill degree, the Ang and Tan split method is the worst here. However, the probably more interesting values in this case are those referring to the overlapping nodes.



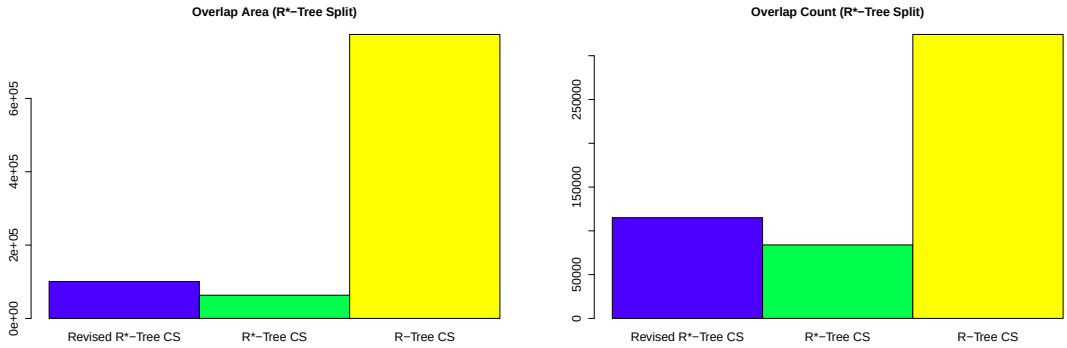
(a) Gazetteer Overlap Area (lower is better)

(b) Gazetteer Overlap Count (lower is better)

Figure 5.46: Gazetteer Overlap Area and Count (lower is better)

The values for overlap count and area can be depicted from figure 5.46. It is easy to see that for Ang and Tan as well as R-Tree split, the R*-Tree choose subtree method works worst for the given dataset. The revised version of R*-Tree is the best choose subtree approach for both versions. R-Tree choose subtree is the second best for this configuration. With respect to overlap count as well as total overlap area, the split version of Ang and Tan seems to perform better than the base R-Tree version. The version of Ang and Tan, however, does not perform so well regarding fill degree and node count. Nevertheless, the best option in all cases (node count, fill degree, overlap area and overlap count) is clearly the R*-Tree split method with one of the respectively inspected methods for selecting a proper subtree to place an item in. Hence, it is probably worthwhile inspecting this method more explicitly in terms of its internal values as they cannot be seen very well in these graphics based on the given dimensions of the other values. Figures 5.47 displays values for the R*-Tree split method regarding overlap count and area. In comparison to figure 5.46, these figures shows enormous differences between the application

[10] `http://download.geonames.org/export/dump/`, accessed 2014-01-14



(a) Gazetteer Overlap Area (R*-Tree Split)

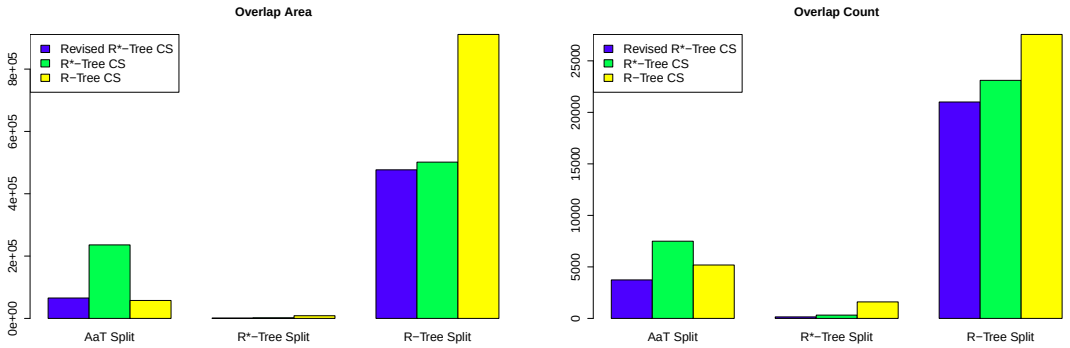
(b) Gazetteer Overlap Count (R*-Tree Split)

Figure 5.47: Values for Parameters (R*-Tree Split)

of the particular methods for choosing a subtree to place an item in. These figures show that the combination of R*-Tree split and choose subtree method outperforms the remaining ones. Especially the base R-Tree linear choose subtree method does not perform similarly efficient as the R*-Tree base and revised methods. Hence, for the Gazetteer dataset, the combination of R*-Tree split and choose subtree performs best as to the criterion of overlap count and area. Concerning fill degree and node count, all combinations supply similar values. That is also true for the other two datasets inspected. Thus, these two parameters are ignored for the remaining analysis here.
The four subfigures illustrated in figure 5.48 show individual results for the other two datasets examined (Reuters and Wikipedia). Only overlap area and overlap count are inspected here, as fill degree and node count do not play an important role in this analysis. The values existing for overlap area and count justify the analysis given for the Gazetteer dataset. It can easily be seen that the splitting methods of Ang and Tan as well as the linear R-Tree method do not perform comparably well as the R*-Tree splitting method. Hence, the focus relative to minimising the overlap is again on the individual values of the R*-Tree splitting method.
Figure 5.49 displays bar plots for R*-Tree split values with respect to the three inspected methods for choosing a proper subtree to place an item in. These graphics show that inside the selected R*-Tree split, the R-Tree linear choose subtree method performs worst regarding overlap area and count for both inspected corpora. The other two methods do not differ as much from each other. Concerning the Wikipedia corpus, the value for overlap area is nearly the same as for the R*-Tree and the revised version whereas the amount of overlaps is slightly lower for the base R*-Tree version. Reuters corpus shows marginally better values for both, overlap area and count, using the revised version of the R*-Tree. Hence, both methods perform comparably well for both corpora.

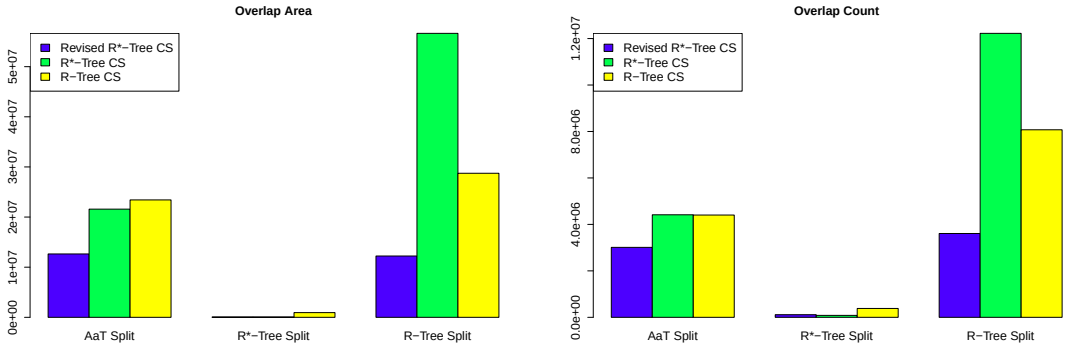
5.2.3.3 Selected Solutions

The basic assumption in this optimisation iteration is that reducing the amount of overlaps in R-Tree elements also reduces the number of calls to the phase of generating lists for fitting elements. The overlap behaviour of the R-Tree can be manipulated by switching between methods for choosing a subtree where to place a particular new R-Tree entry as well as the distribution of elements to new nodes when a split occurs. For this purpose, different combinations of distribution possibilities are evaluated in order to find the best pair of approaches. The baseline for this evaluation is the R-Tree linear / R-Tree linear combination as it is used initially by the hybrid index for R-Tree construction.
The evaluation of the three split methods, R-Tree linear, Ang and Tan and R*-Tree, in combination with the three inspected methods for choosing a subtree, R-Tree linear, R*-Tree and revised R*-Tree shows the following results regarding overlap behaviour:



(a) Reuters Overlap Area

(b) Reuters Overlap Count



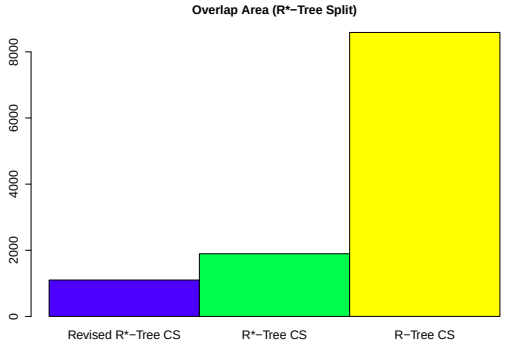
(c) Wikipedia Overlap Area

(d) Wikipedia Overlap Count

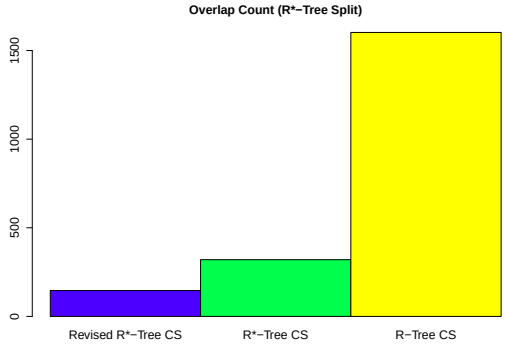
Figure 5.48: Summarised Overlap Values for Reuters and Wikipedia

- Ang and Tan split method is superior to R-Tree linear split,
- R*-Tree split performs better than Ang and Tan split,
- R-Tree linear choose subtree shows the poorest behaviour,
- Revised R*-Tree choose subtree performs best for all splitting methods except R*-Tree split,
- R*-Tree split in combination with R*-Tree choose subtree or the revised version performs best for all corpora,
- Depending on the corpora internals, R*-Tree or revised R*-Tree perform best,
- It is not always obvious which of the two best performing choose subtree approaches to select.

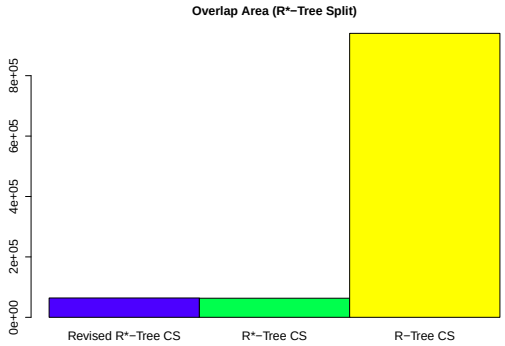
Therefore, as R*-Tree and revised choose subtree show a similar performance regarding overlaps, the decision may be taken arbitrarily. However, it is probably worthwhile investigating the concrete corpus and taking individual measurements in order to depict the best combination for the respective corpus.
As the evaluations are mainly executed on the Wikipedia corpus, which shows best results for R*-Tree split and choose subtree implementations, this combination is chosen as selected solution for this iteration. It must be kept in mind that investigating the corpus behaviour or the expected one could be seen as a preliminary step for setting up the best combination for the use in the hybrid index. Besides the effects awaited on list generation, also the R-Tree expansion phase used to distribute new point values to the R-Tree may be compared in the evaluation of this optimisation iteration.



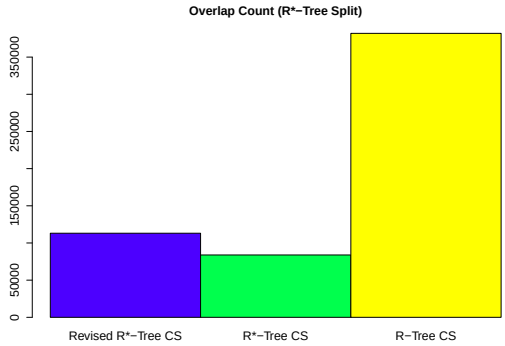
(a) Reuters Overlap Area (R*-Tree split)



(b) Reuters Overlap Count (R*-Tree split)



(c) Wikipedia Overlap Area (R*-Tree split)



(d) Wikipedia Overlap Count (R*-Tree split)

Figure 5.49: Summarised Overlap Values for Reuters and Wikipedia (R*-Tree split)

5.2.3.4 Evaluation

The evaluation validates the selected results against the expected outcome of the iteration. The parameter settings for the post-test run can be found in table 5.11. The most notable modifications are the change in choose subtree and split method for the R-Tree (coloured grey). This test run is executed with R*-Tree split and R*-Tree choose subtree methods. The remaining settings stay the same as in the pre-tests. Again, the main evaluation is split in two parts: general evaluation for analysing the respective phases and individual evaluation for the particular phases which have turned out to be inefficient based on the measurements in the paths analyser tool.

5.2.3.4.1 General Evaluation The effects for the entire optimisation can be seen in a comparison of the function `add` which is responsible for inserting one item. They are displayed in figure 5.50. These effects are not as tremendous as in the first iteration, but a slight improvement of the average runtime can be depicted.
A total overview of the respectively inspected phases can be seen in figure 5.51.
Minor effects on the total runtime can be demonstrated for `rtreeExpand` and `generateList` phase. Although, the two other (actually not inspected) phases of inserting elements to the initial inverted index and the secondary inverted index manipulation have a slightly higher average duration. Reducing the average duration of generating the lists of fitting elements as well as R-Tree expansion is a desired side effect of this optimisation iteration. Generally, the iteration can be seen as successfully completed because the total insertion operation is optimised regarding reorganisation performance with special respect to the list generation as well as R-Tree expansion.

Parameter	Value	Explanation
HLimit	200	Artificial Upper Bound
Element Count	5	Amount of Elements per R-Tree node
Document Count	1989	Amount of Documents to be inserted
Dataset	Wikipedia	Dataset to be processed (Wikipedia sorted by ids)
Split	**R*-Tree Split**	R-Tree splitting method
Choose Subtree	**R*-Tree Choose Subtree**	Method for Selecting a proper subtree to place an element in

Table 5.11: Settings for the Post-Tests Test Suite Run

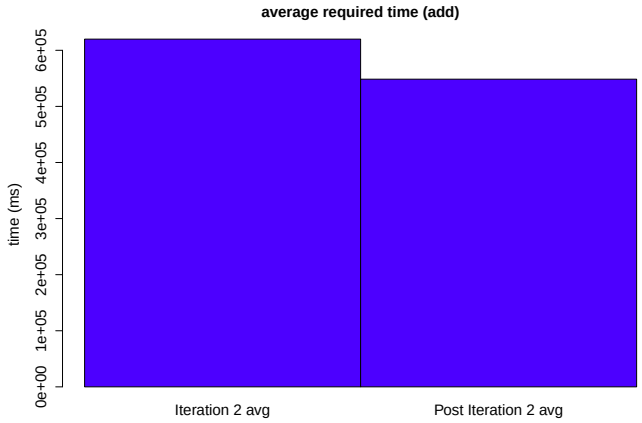


Figure 5.50: Add Function Comparison

5.2.3.4.2 Individual Evaluation The detailed evaluation of the respective phases just as the selected phases generated by the paths analyser tool follows in this subsection.

The main phases to be inspected in this iteration are the function to generate lists and the R-Tree expansion phase. Figure 5.52 shows the average runtime of these phases. It is possible to reduce the list generation average runtime slightly as an outcome of the optimisation. The resulting average runtime of this phase is $\approx 15.26\%$ lower than the initial run. The respective time frame for the R-Tree expansion phase could also be decreased and the duration of this phase may be lowered by an amount of $\approx 27.25\%$.

Function	Duration Pre	Amount Pre	Duration Post	Amount Post
rtreeExpand	3, 108, 590.6210	8, 027	2, 261, 495.1490	8, 027
generateList	7, 335, 648.8770	116, 455	5, 540, 625.6270	103, 796

Table 5.12: Durations and Amounts of Calls in Pre- and Post-Tests

The detailed values for the two phases of investigation can be seen in table 5.12. The R-Tree expansion functionality is executed once per point value given in the document set. Hence, it is obvious that the amount of calls to this function has not declined for this optimisation. The internal execution of the function first screens for the presence of the respective item inside the R-Tree and, if it is not existent, places the item inside the tree. Hence, this phase makes extensive use of the two changed methodologies of choosing a proper subtree and subsequent split after placing the particular item with the aim to reorganise the R-Tree in order to fit to the given requirements of balancing and maximum amount of elements to be stored. It also

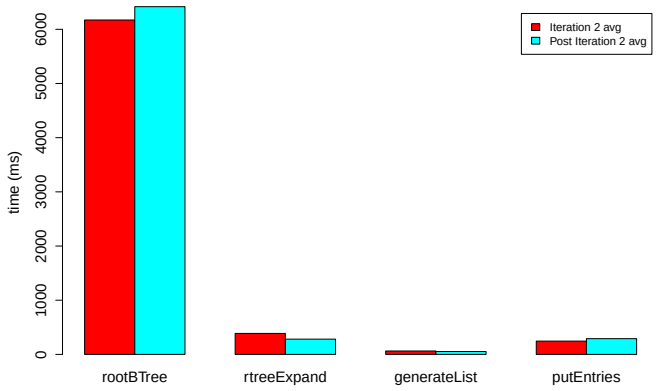
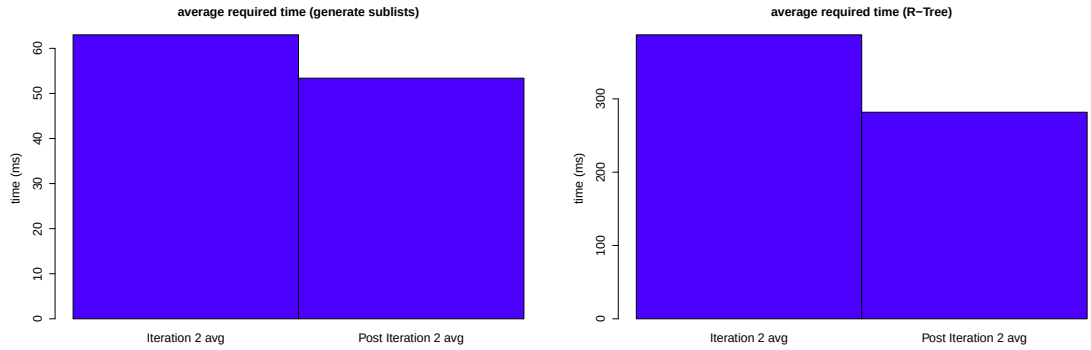


Figure 5.51: Total Overview of Iteration 2 Comparison



(a) Generate Function Comparison

(b) R-Tree Modification Function Comparison

Figure 5.52: Individual Phases Comparison

becomes obvious that the internal implementations of these functions tend to be more efficient than the ones used before. Besides this fact, it is also true that when trying to find one particular element inside the R-Tree and with not many overlaps existing, a lower amount of nodes must be examined which do not contribute to the search result. Thus, refinding an item inside the R-Tree tends to be more efficient. Therefore, as the amount of calls to the R-Tree expansion stays the same as in the initial pre-test, the performance gain in total is the same as the average gain of $\approx 27.25\%$.
The probably more interesting phase is the generation of the lists valid for the given R-Tree element. This phase is now evaluated regarding pre- and post-test. First, it can easily be seen in table 5.12 that the amount of calls lowers by 10.87%. This is mainly an effect of the improved distribution of items inside the R-Tree. It is obvious that, if the respective R-Tree elements do not share much common space, there is no need to investigate (or at least more seldom) overlapping areas. Therefore, regarding the place for an item to be found during the distribution run, there are enhanced pruning mechanisms so that also less items must be examined in each run of the list generation algorithm. This lowered amount of calls to this function already displays the effects of the improved distribution of items using the R*-Tree algorithms. Besides this effect, figure 5.52a also shows that the average time required for one individual execution of the phase lowers by $\approx 15.26\%$, too. This is a result of the improved pruning effects introduced by the use of a different distribution of items inside the R-Tree as well. More items can be eliminated from the list of valid items for the respective subtree if better pruning is applied. Therefore, the effect of building distributions with fewer overlaps by using adopted versions of splitting a particular overfull node and choosing a proper subtree where an item shall be placed,

improves, on the one hand, the amount of times the list generation is executed and, on the other the, average duration per execution.

Function	Duration Pre	Amount Pre	Duration Post	Amount Post
pointEquals	3, 861, 800.545	288, 136, 287	2, 280, 833.499	169, 657, 648
rangeContains	1, 327, 086.688	100, 160, 525	1, 512, 369.446	117, 019, 660

Table 5.13: Pre- and Post-Test Comparison of Predicate Checks

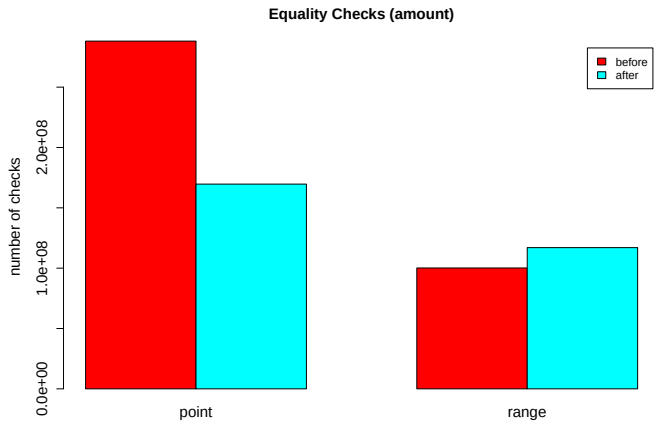


Figure 5.53: Comparison Functions before and after Optimisation

A detailed comparison of values using a pre-post test evaluation can be seen in table 5.13. Figure 5.53 shows the same values in a bar plot representation. Obviously, above all the amount of comparisons carried out for points has lowered significantly. This indicates that the number of leaf nodes visited also lowers. As it is estimated that the quantity of nodes per level rises approximately exponentially, checks in leaf nodes without contributing results, which means that nothing is done at these leaf elements, must be targeted as most nodes exist at leaf level. The amount of comparisons there could be lowered by $\approx 41.12\%$. The comparisons on ranges, which means in inner nodes, rises by 16.83%. However, the total number of comparisons is reduced by $\approx 26.17\%$ leading to the improved runtime behaviour.

5.2.3.5 Summary

This iteration is driven by the impression that a distribution free of overlaps leads to a performance boost due to a lowered effort of the distribution function. The basic hypothesis is that reducing the overlap also lessens the amount of checks for items during list generation. An empirical study using nine different potential candidate combinations of distribution possibilities is executed and shows that the best combinations are:

- R*-Tree Split with R*-Tree choose subtree, or
- R*-Tree Split in combination with revised R*-Tree choose subtree

As, depending on the corpus, any of them may be chosen, the decision is taken to set up R*-Tree split with R*-Tree choose subtree and execute the post-tests. They show an improvement of 11.52% regarding the runtime in the inspected corpus. This is mainly due to the improvements of the runtime behaviour of the list generation phase resulting from the improved properties of the R-Tree overlap that depends on the proper setup of distribution methods.

Besides these changes, the search procedure must be looked at additionally. The changes performed in this iteration do not influence the search procedure. With the internal distribution of elements affected in this optimisation, the process of refinding an item is the same as before and thus, the search functionality need not be adopted.

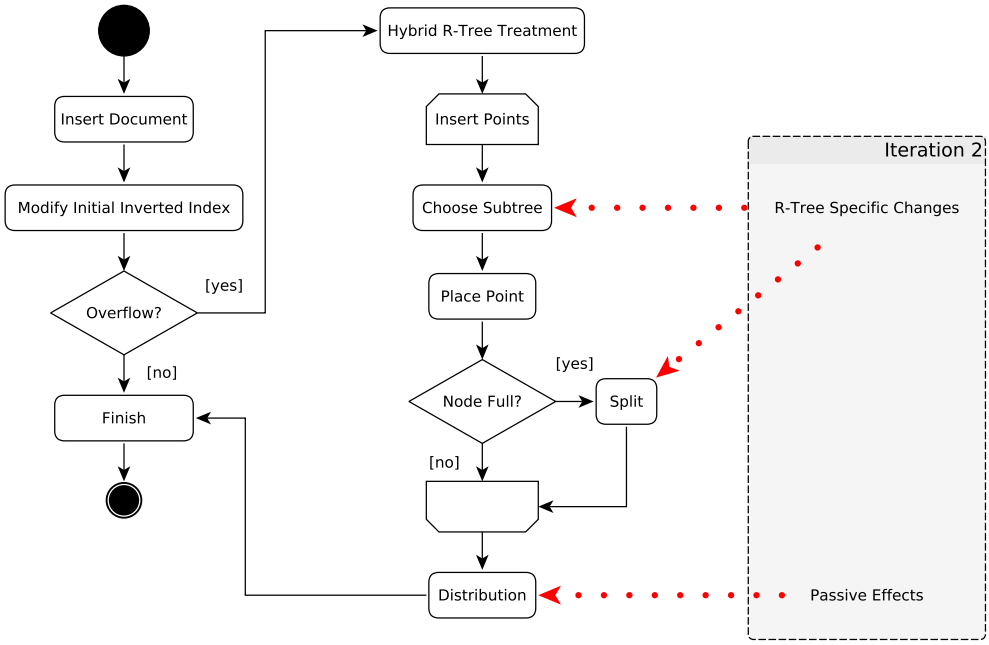


Figure 5.54: Activity Diagram of Changes Performed in Iteration 2

The changes performed in iteration 2 are presented in figure 5.54. The main target of this iteration is the R-Tree specific adoption which mainly refer to the distribution of point values inside the hybrid R-Tree variant. Especially the algorithm for choosing a proper subtree for the placement of points as well as the splitting mechanism are investigated. Hence, there are direct influences on these two parts of the algorithm where the R-Tree modification is carried out. Based on the measurement results, it can also be seen that side-effects exist based on the improved algorithm variants which refer to the distribution of the elements throughout the R-Tree and secondary inverted index manipulation. They are marked as passive effects in figure 5.54.

5.2.4 Inverted Index and Efficient Storage of Entries

The hybrid index structure to be improved in this thesis relies on a heavy use of inverted index structures. They are utilised in an initial part which maps from a particular term to a term id or also stores references to the document heap for the retrieval of documents, directly. Besides this initial inverted index, there are also secondary inverted index structures. They can be found at R-Tree leaf elements storing the final assignments of terms to document heap references. Especially in this part of the structure, a lot of work is done in order to store the data properly. The first two approaches primarily optimised the distribution of elements inside the R-Tree using caches or the R-Tree as such by applying adopted versions of the R-Tree variants. Hence, it is now time to look closer at the inverted index structures as they are also an important part inside the reorganisation procedures.
Besides the work on the inverted index structures, again, considerations must be made to find out about the proper storage of elements somewhere in the hybrid index structure. The current storage mechanism of individual items seems to be still inefficient regarding the retrieval and storage of the respective items. Thus, an investigation of the persistence and the inverted index structures is carried out now.

Iteration 3 starts with a given set of pre-tests executed on basis of the results from the previous iteration. The actual settings for executing the test suite are directly taken from iteration 2.

Parameter	Value	Explanation
HLimit	200	Artificial Upper Bound
Element Count	5	Amount of Elements per R-Tree node
Document Count	1989	Amount of Documents to be inserted
Dataset	Wikipedia	Dataset to be processed (Wikipedia sorted by ids)
Split	R*-Tree Split	R-Tree splitting method
Choose Subtree	R*-Tree Choose Subtree	Method for Selecting a proper subtree to place an element in

Table 5.14: Settings for the Pre-Tests Test Suite Run

Hence, the chart of parameters can be found in table 5.14. Actually, the output data from the post-tests of iteration 2 are utilised directly as input values for this iteration. Again, 78 repetitions of the test suite execution are done in order to gather data for the current iteration information. In this case, on grounds of the outcome of iteration 2, R*-Tree methods for splitting and selecting a proper subtree to place point values in are used. As seen in iteration 2, the outcome of the already finished iterations results in the currently arising potential issues.

5.2.4.1 Basic Observations

An overview of the absolute duration of the respective phases in this iteration is displayed in figure 5.55.
This graphic does not show a clear tendency. Compared to the previous iteration, the list generation phase has improved significantly. Again, the root B-tree is not very substantial. It was also noticed in the previous iteration that the phase of putting particular entries to the secondary inverted index structures is of similar influence as the list generation. Yet, due to the optimizations executed in the previous iteration by changing the methodology of splitting nodes or choosing proper subtrees to put the respective point values to, the insertion of items to secondary inverted indices has not changed. Thus, based on the improvements of the runtimes

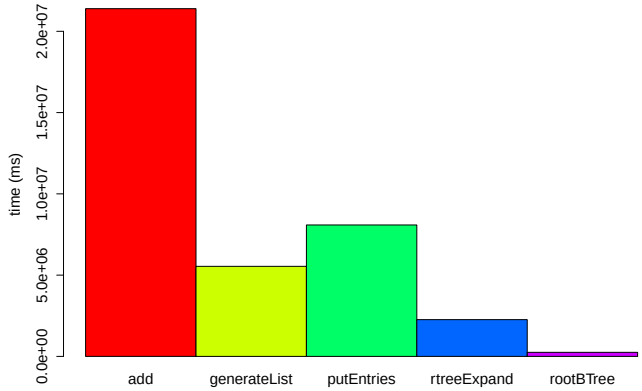


Figure 5.55: Duration of Phases

of the two phases (`generateList` and `rtreeExpand`), this phase shows up to be the worst in this measurement run.
The most influential executions require ≈ 25.9% (`generateList`) and ≈ 37.8% (`putEntries`) of the entire runtime. The remaining phases only need ≈ 10% and less of the time required by the entire reorganisation process. It must be noted that the leftover of the time is spent in additional functions caring mainly about management functionalities.

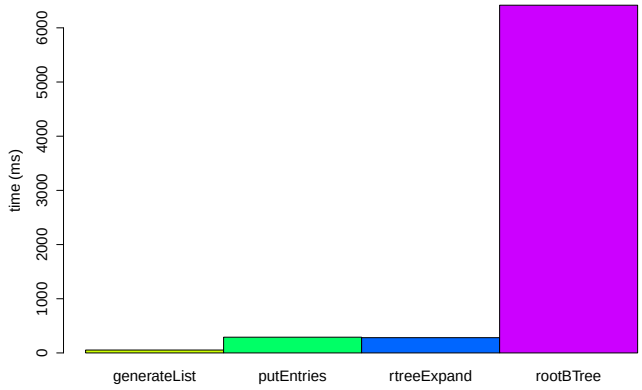


Figure 5.56: Average Runtime per Phase

The average runtimes of the respective phases are shown in figure 5.56. The initial inverted index manipulation phase is the one with the largest average duration. Basically, this originates from the fact that this phase is executed exactly once per reorganisation operation. As it can be seen in figure 5.55, the total runtime is not very influential in this case instead. The phase of expanding the R-Tree as well as the modification of the secondary inverted index structures have a comparable average runtime behaviour. The best average runtime of the inspected phases shows the method for generating the lists. This is a desired effect because the last two optimisation iterations had the focus to improve exactly the behaviour of this method.
Hence, regarding a comparison of total and average runtime, the putting phase is most disadvantageous for the total runtime. The average runtime shows worst effects for adding items to the initial inverted index. In the second place are `rtreeExpand` and `putEntries`. The putting phase is comparable to the initial inverted index phase. Basically, the two phases perform similar tasks because they both modify an inverted index. The secondary inverted index

Parameter	Value
Local Threshold	0.1
Global Threshold	0.2
Maximum Path Steps	3

Table 5.15: Parameter Values for Iteration 3

consists of a mapping between term id and document pointer whereas the initial inverted index does the same for terms instead of numerical ids.
The individual analysis is executed by using the paths analyser tool with the parameters given in table 5.15. The local threshold value is quite low. The same holds for the global threshold and the maximum path steps parameter is fixed to a value of three. Thus, merely relatively short paths are taken into account which do not differ much (local threshold). Only a small amount of program execution paths are pruned because the global threshold is quite low resulting in not many pruning effects during graph traversal. The basic results show similar values as the general observations.

Function	Duration	Amount
putEntries	8,085,553.142	27,836
generateListForElement	5,540,625.627	103,796

Table 5.16: Overview of Results from the Individual Analysis

The essential outcome of the individual analysis run using the paths analyser tool for iteration 3 can be seen in table 5.16. This chart displays the two methods identified in the general analysis, as well. Compared to each other, it can be stated that `generateList` is called ≈ 3.73 times more than `putEntries` requiring only $\approx 68.53\%$ of the runtime. Hence, as already seen in the general analysis, the total duration as well as the average runtime of `putEntries` is significantly worse than the one of `generateList`.
It is probably worthwhile investigating the detailed circumstances for `putEntries` to be executed by the reorganisation algorithm and the effects caused by one activation. This algorithm has the task of inserting the entries to the final destination inside the hybrid index structure. Therefore, it is executed when a leaf node is hit during the recursive distribution of the items to the hybrid index. At each point entry of the leaf elements, a list of items is calculated for the respective point. At each point, a list of term ids to documents is generated which contain the term represented by the id and refer to the point currently inspected. The algorithm takes each of these assignments and inserts them into a B-Tree representing the inverted index at this particular R-Tree leaf element. Thus, this final passage distributes the term ids to the point values inside the R-Tree and hence performs the intersection of terms and points inside the R-Tree. Consequently, for each point contained in the set of documents overflown in the currently inspected reorganisation run, the algorithm is executed once performing (at least) one insertion at one particular secondary inverted index. The effort spent in this phase depends on the number of terms to be assigned per point value occurring together in the particular documents. Additionally, if a lot of points values are affected in one reorganisation, also a lot of modifications of the secondary inverted index structures must be carried out.
In detail, this method takes each single term id, chooses a proper subtree to insert the element at starting from the root node and reorganises the structure of the B-Tree after the insertion. An overview of the respective B-Tree manipulation phases is shown in figure 5.57. It displays that the retrieval of the root node of the B-Tree is the main time consumer in this phase followed by the `putAndAdjust` phase which searches a proper place to insert the corresponding item from a particular start node element, adds it at the particular place and reorganises the tree. The other two methods (`chooseSubtree` and `findPlace`) do not have similar effects as the first two. It must be noted that each occurrence of each term id is inserted individually in this phase. Besides this observation, it is probably worthwhile investigating the, still inefficient, phase of generating the lists valid for subtree elements. The comparisons carried out inside the list generation are the dominant methods regarding runtime.

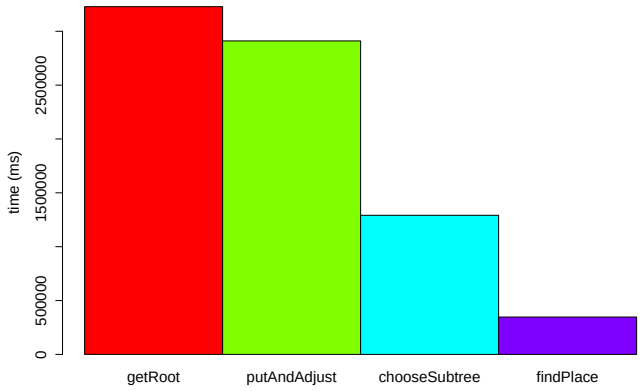


Figure 5.57: Inverted Index Manipulation Phases

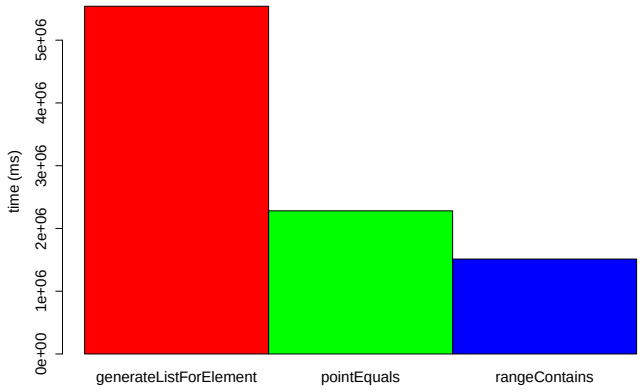


Figure 5.58: Comparison Functions and List Generation

The duration of the comparison functions and list generation for iteration 3 is displayed in 5.58. This image shows that the most runtime for list generation is spent on comparisons of points or checks for containments of points in ranges, respectively. Thus, an investigation of the internals of these functions is done, now.

Function	Duration	Amount
BitList.equals	422, 834.717	250, 784, 242
RTreeContent.equals	8, 385.738	503, 280, 412

Table 5.17: Individual Comparisons

Chart 5.17 shows the details of the function comparing two R-Tree content elements with respect to points. This is, currently, implemented as object-wise comparisons. It can be seen that the comparison of bitlists, which is not at all taken into account in this stage of the insertion algorithm, consumes the most time. The actual comparison is executed in `RTreeContent.equals` which then checks the point equality or range containment. It can easily be seen that it is executed quite often only requiring a very small amount of time. The additional bitlist comparison relatively consumes a much longer period of time without contributing at all to the actual comparison (as it is not taken into account at this stage).

5.2.4.2 Solution Alternatives

The basic observations given in the previous sections showed two essential problems at the current stage of the optimizations:

1. Executions of comparisons in inner R-Tree elements and
2. inverted index manipulations

Therefore, an investigation of the respective parts is carried out, now.

5.2.4.2.1 Execution of Comparisons It is necessary to investigate the actual process of executing the comparisons to be able to introduce a performance boost in this functionality. The comparisons are carried out based on the data stored in the respective nodes where the data are stored in a serialised way.

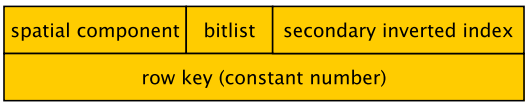


Figure 5.59: Tuple Construction for the Hybrid R-Tree

Figure 5.59 shows the basic setup of one tuple of, e.g., the R-Tree (see section 3.2.1). In the current state, this tuple is stored inside one `Value` object as part of a `SearchRow`. This leads to the fact that for each comparison of, for instance, two point values, the tuple must be loaded and deserialized into an object of type `RTreeContent` which stores the given information as fields internally. The separate parts (spatial component, bitlist and secondary inverted index reference) also must be deserialized to be composed to the actual object again. This procedure seems to be ineffective because if, for example, solely the spatial component is to be checked, all the components of the content object are deserialized even if they are not required for the present comparison. Comparing only the spatial component is not an issue because the deserialisation process could stop having read the bytes from the spatial part. The other parts, instead, are more complex. Skipping pieces from the byte stream is not as easy as stopping after reading a predefined portion in the beginning of the stream because data are presented sequentially. Hence, another solution must be found for this issue. This (de-)serialisation process is executed for each portion of the stored objects in all parts (initial inverted index, hybrid R-Tree and secondary inverted index). However, the R-Tree stores most of the information as the initial inverted index only persists the term to document heap reference and the secondary inverted index saves similar information (term id to document heap reference). Yet, for the hybrid R-Tree, extracting specific information from the stored tuples is essential because during descending the tree in the insertion part, mainly the spatial components are investigated. During the adjustment of the R-Tree, especially the bitlists are subject of investigation. In the secondary inverted index manipulation part, the reference to its root node is modified.
For this reason, a possibility of addressing the specific parts of tuples in a more direct way is aimed at. Thus, the decision is taken to split up the respective storage areas of all affected parts into more separated elements. The basic construction of a `SearchRow` can be depicted from figure 5.60. The `SearchRow` is set up of a row key which either references an entry in the original user table or can also be applied to point to subtrees in the case of tree oriented storage. The other part of the `SearchRow` is an array of `Value` objects. They may be used to store all kinds of elements utilised inside the database. Representations for all native data types (`ValueInt`, `ValueDouble`, ...) as well as for complex ones (`ValueJavaObject`, `ValueBytes`) are present inside the core of the H2 database. In the current state of the implementation, a serialised version of the data objects is stored in `ValueBytes` structures.

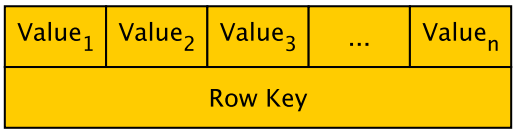


Figure 5.60: `SearchRow` Basic Setup

The data are serialised manually to byte arrays from which they are read when they are required. This procedure is cached by using a LRU approach, as described in iteration 1 (see paragraph 5.2.2.3.2).
The decision for this optimisation iteration is taken to split the original data objects like the `RTreeContent` object to its individual storage parts. The original tuple construction contained all the data elements in individual fields of this kind of objects. The new approach uses multiple `Value` objects to store the respective data in. That means that the original `RTreeContent` is saved as one object with individual fields representing the spatial component, the bitlist and the reference to a secondary inverted index as one `ValueBytes` object. The new version of storage splits these data objects in three individual storage domains. Hence, the `RTreeContent` is decomposed into its three individual parts each of which is stored in a separate `Value` object. It must be noted that also the way of representing the objects must consequently be changed. The original way is to store all values related to one column in one `Value` object of a `SearchRow`. The new persistence approach is allowed to store more than one value referring to one column inside one `SearchRow` object.
The desired effect in this case is that by separating the `Value` objects in multiple individual ones helps enhancing the comparison and modification speed because the particular values can be loaded more directly from storage pages. Based on this change, also the storage types and (de-)serialisation process needs to be changed in order to retrieve or be able to save the respective entries properly.

5.2.4.2.2 (Secondary) Inverted Index The second, probably more important, change to overcome the current weakness in this iteration is the modification in the secondary inverted index structures. By improving their performance, the one of the initial inverted index should be increased, too. Hence, the entire inverted index structures are target of investigation now. During this iteration, a model using a mathematical description of the respective storage part is created. This model is described in chapter 4. It is built in order to derive specific parts and coherences formally.
For optimisation approaches of the inverted index structures used in the given hybrid index, literature studies showed a very essential result of storage alternatives. There are multiple alternatives for managing modifications of inverted index structures (see i.e. [11, 21, 32]) which try to optimise the reorganisation behaviour of inverted index structures based on statistical pre-computations or adapted algorithms.
A detailed inspection of the storage procedure utilised at present is inevitable at this stage. The currently present algorithm stores the inverted lists directly inside the B-Tree leaf nodes. That means that if there are a lot of occurrences of one particular term inside the document corpus, also a lot of references of this term to the documents it is contained in, the same amount of entries pointing from the term to each of the documents is stored inside (at least) one leaf node. Another alternative to this storage procedure is to store simply the directory (individual terms) in the nodes of the B-Tree and save a further inverted list with the postings in which the term occurs in an additional storage structure. The first option probably shows performance issues if a lot of terms are included in many documents. If a lot of terms occur frequently in the corpus, the branching factor of the B-Tree lowers and thus the amount of levels is much higher when the list of postings is quite long. On the other hand, if the lists are short and the postings were to be stored in external pages, an additional page must be loaded for every term even if the number of occurrences of the term inside the corpus is only 1. Hence, the first option seems to be superior

if the posting list is small. The other option favours rather long lists as the tree stays more dense in this case.
Therefore, an additional option is introduced using a "hybrid" storage option. This option takes into account a different treatment of frequent and infrequent terms. Basically, the considerations depend on Zipf's Law. The third option relies on the fact that if there is more than one pre-defined amount of references to one particular term, it is rational to switch storage strategies. That is why, in this strategy, first the entries are stored directly inside the B-Tree. If there are so many elements stored in a B-Tree leaf node that the required storage is greater than the available memory space, all but one references to exactly this entry are removed from the B-Tree and an inverted page is constructed saving the posting list externally. Hence, this strategy tries to benefit from the direct storage of pointers in the case of terms with low frequencies and from external storage of the postings list for terms occurring very often. The basic situation for this consideration is that if the memory space required for one term exceeds the storage of one B-Tree node, an additional page must be loaded. Thus, when moving all but one entries of this term from the B-Tree to the postings list, at most the same number of pages must be loaded. Besides this fact, due to the removal of the remaining references, the B-Tree will tend to have less leaf nodes, which means that the density gets higher and the B-Trees tend to be smaller in terms of levels.
An empirical study is executed in order to verify these presumptions. The input data taken for the experiments consist of the Wikipedia dump (see section 5.1.3.2). For this study, only the textual part is used as an input. An inverted index implementation is applied inside the database using a B-Tree as directory structure. It supports three different storage strategies to be evaluated in order to figure out the optimised insertion strategy. Besides observing the insertion, also the query performance of the respective strategies are monitored.
The three strategies under test are:

1. LEAF storing all entries as well as the entire posting list in the leaf nodes of the B-Tree,
2. EXTRA saving the posting list in additional structures (called inverted pages) and
3. HYBRID performing a mixed strategy between leaf and extra strategy by using memory allocation checks, as described in advance.

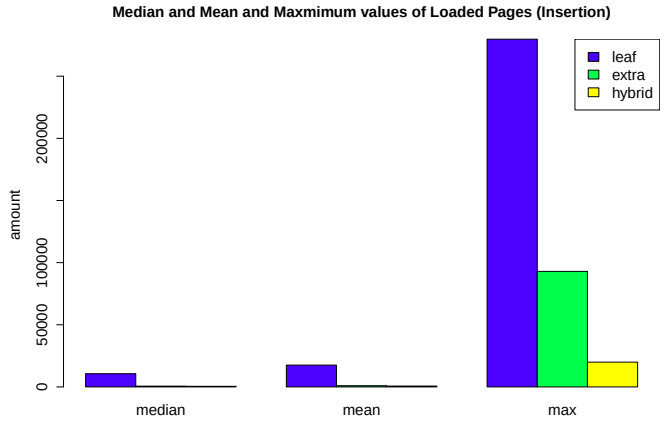


Figure 5.61: Median, Mean and Maximum Number of Loaded Nodes during Insertions

The median, average and maximum amount of loaded nodes during the insertions is displayed in illustration 5.61. This figure shows that, at least regarding the maximum amount of loaded nodes, the option to store all data directly inside the leaf nodes performs worst. As loading one particular page is directly related to a certain amount of time (usually in millisecond resolution), the need to load this significantly higher amount of pages relates to a much higher time. With

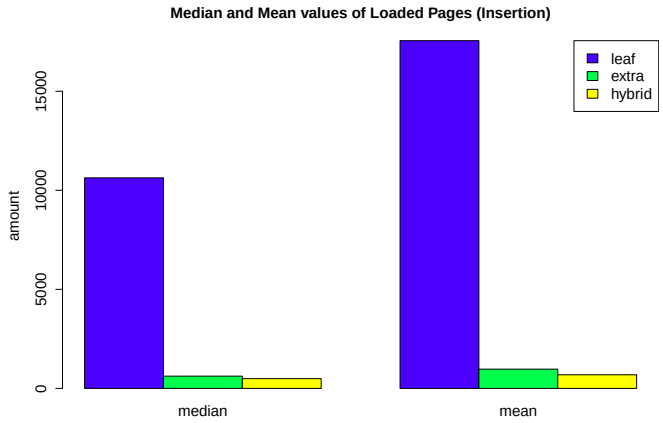


Figure 5.62: Median and Mean Number of Loaded Pages during Insertions

respect to the maximum number of loaded pages, the leaf storage option is the second best whereas the hybrid approach performs best in this case. For inspecting the median and mean values, a separate figure is required because, based on the fitted values in figure 5.61, the differences cannot be depicted in detail.
Figure 5.62 shows an overview of the median and mean values for cumulative sums of loaded pages during the insertion runs. It can be seen that the leaf storage method is clearly worst regarding these two parameters. Similar effects as in the maximum analysis can also be depicted from these figures. As to the total amount of loaded pages the best method is the hybrid storage strategy which decides whether the leaf or the extra strategy must be applied, depending on the term frequency. The worst option is the direct storage inside the leaf nodes of the B-Tree. In total, as regards the amount of loaded pages during insertion runs, the best option with respect to all parameters is the hybrid strategy.

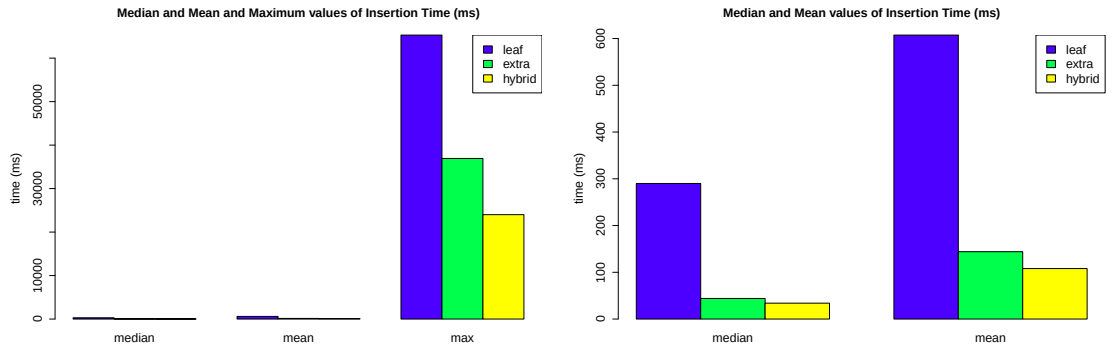


Figure 5.63: Maximum, Median and Average Times Required for Insertion

The numbers for insertion times, measured in milliseconds can be depicted from figure 5.63. It is easy to see that the quantities given in the both subfigures of figure 5.63 correlate with the amount of loaded nodes during the insertion (see figure 5.62). Thus, it has been shown that the hybrid strategy is definitely the best regarding insertion performance both with reference to insertion time and as well as to the amount of loaded pages.
Yet, not only the insertion performance should be monitored in this study. There is a certain likelihood that the best strategy during insertions performs worst with respect to search queries. Therefore, it is inevitable to look at the query performance, as well.
Illustration 5.64 depicts the summarised numbers for the query performance of the respective inverted index strategies. This figure again shows that the leaf storage option is obviously the worst, also as to retrieval throughput. The remaining two options perform similarly efficiently.

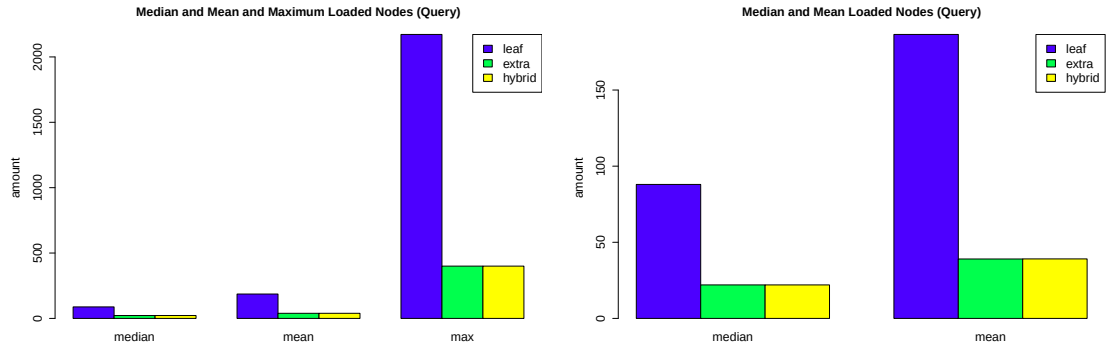


Figure 5.64: Maximum, Median and Average Number of Loaded Pages for Queries

Hence, it can be seen that the hybrid storage option does not only perform best regarding insertion time and loaded pages but also with respect to searches. The extra storage strategy performs comparably well in concerns of retrieval. Yet, it is not as efficient as the hybrid option when inserting items.
It must be noted that the results have been verified with the Reuters corpus with similar outcomes.

5.2.4.3 Selected Solutions

The solution alternatives for the given performance issues are presented in the previous subsection. Actually, for changing the storage of elements, only one solution is presented, which however is essential for the changes, done here. The entire storage of all structures must be altered in order to avoid the (de-)serialisations to the respectively composed objects by storing the individual parts of the particular structures individually inside the `Value` array of a `SearchRow`.
This change refers to all storage structures involved in the reorganisation algorithms. That means that it is necessary to revise all structures. This involves the initial inverted index, the hybrid R-Tree as well as the secondary inverted index. The detailed changes for this section are that the initial inverted index does no more store the data in one single object but uses three individual `Value` objects for the storage of the term, the term id assigned, if it is already stored in the secondary inverted index, and the reference to the document heap. The hybrid R-Tree is revised to store three `Value` objects in terms of the spatial component (range or point), the bitlist to indicate the presence or absence of a certain term in the particular subtree, as well as the reference to the secondary inverted index which is mainly important in leaf nodes of the hybrid R-Tree. The secondary inverted index structures now point from an integer value, no longer encapsulated in a so-called `InnerBTreeContent`, to a document heap reference. This affects changes in all algorithms and thus is a major modification.
The second alteration taken as solution for this optimisation iteration is the revision of the inverted index structures. This, again, affects more than one structure. On the one hand, the initial inverted index, on the other, the secondary inverted index structures must be reworked. However, as the inverted index manipulation is encapsulated in a module, it is not necessary to modify a lot as the interface methods can be used to properly distribute the items. Due to the results of the empirical study described in paragraph 5.2.4.2.2, it was decided to realise the HYBRID strategy inside these structures. These changes are introduced based on the empirical study which showed a clear performance boost of this strategy in both, reorganisation and retrieval, algorithms.

5.2.4.4 Evaluation

The post-tests for the third iteration start with exactly the same parameter settings as the pre-tests because they are not subject of investigation for this test iteration.

Parameter	Value	Explanation
HLimit	200	Artificial Upper Bound
Element Count	5	Amount of Elements per R-Tree node
Document Count	1989	Amount of Documents to be inserted
Dataset	Wikipedia	Dataset to be processed (Wikipedia sorted by ids)
Split	R*-Tree Split	R-Tree splitting method
Choose Subtree	R*-Tree Choose Subtree	Method for Selecting a proper subtree to place an element in

Table 5.18: Settings for the Pre-tests Test Suite Run

The chart of settings can be seen in table 5.18. The same values are used for the initial pre-tests in the initialisation part of this iteration.

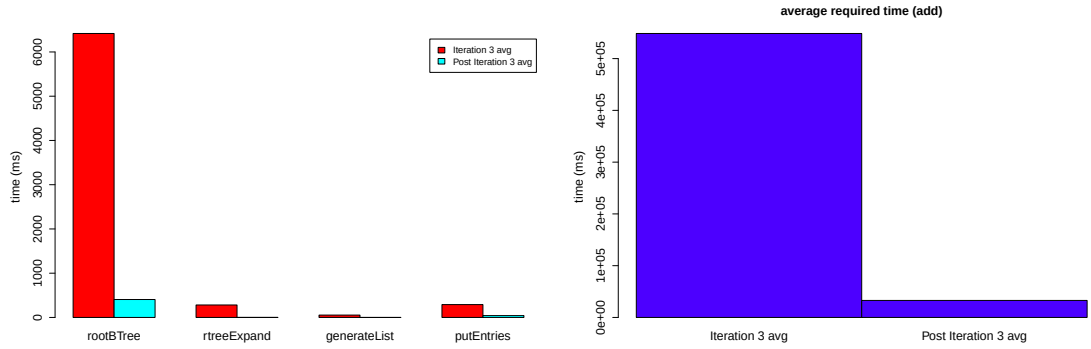


Figure 5.65: Total Overview of Iteration 3 and Add Function Comparison

5.2.4.4.1 General Evaluation Figure 5.65 shows the total overview of the individual phases of the third iteration as well as the detailed comparison of the average runtime of the add function performing the reorganisation of the hybrid index structure.
It can easily be depicted that the effects of this optimisation iteration are tremendous. Unfortunately, it cannot be stated definitely which of the optimizations are responsible for the effects at this stage. At least it is impossible to determine for the total overview of the iteration and the insertion function unambiguously. To highlight the details, the individual evaluation must be inspected which analyses the respective optimizations and their effects in a more precise way. It can be depicted that the average runtimes of all phases could be lowered dramatically based on the given optimizations in this iteration.
The numbers in figure 5.65 (right picture) show tremendous effects on the average runtime of all phases. The most notable change turns out at the initial inverted index reorganisation phase (`rootBtree`) because each call summarises a series of insertions at this stage. The data inserted into the initial inverted index comprise all terms of one particular document. Hence, the average runtime refers to the one of all documents. It could be reduced by a large factor.
The remaining phases most probably profit from the improved implementation of storage access. This is analysed in detail in the following paragraph.

5.2.4.4.2 Individual Evaluation The individual evaluation in this case refers to all phases because the changes in the basic storage mechanisms using separate `Value` objects per respectively composed attribute. Thus, obviously, each phase needs to be inspected as this iteration basically introduces modifications to each structure. Yet, the most important differences are introduced in the inverted index parts of the hybrid index structures. Therefore, the individual evaluation is split in two parts:

1. Storage structure (general) evaluation and
2. Inverted index evaluation

The first evaluation part does not examine the inverted index structure phases (`putEntries` and `rootBtree`) because they are inspected afterwards.

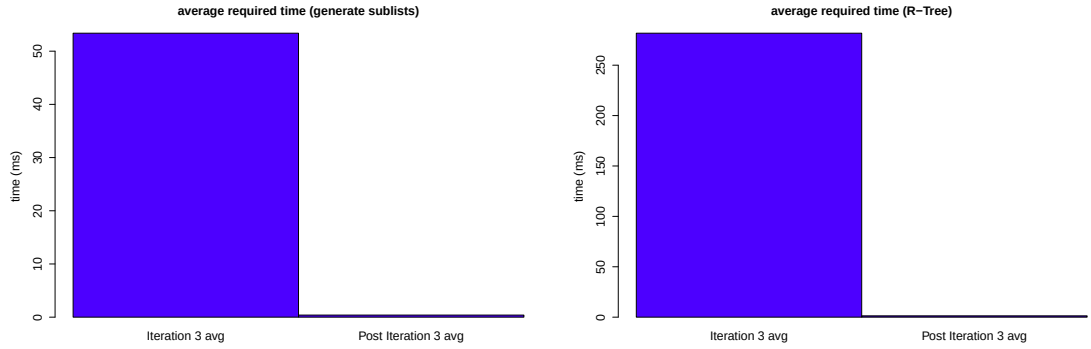


Figure 5.66: List Generation and R-Tree Expansion Comparison

Figure 5.66 displays the average runtime values for the list generation and the R-Tree creation phase. It can be seen that they could be improved by a big order of magnitude. Unfortunately enough, the effects of the loading process cannot be depicted directly from the data as it is not monitored by the application. However, as the storage as well as the retrieval of data to and from database pages and the entire internal storage have entirely changed, it seems to have effects on the efficiency. It is important to say that the management of data is transformed into a more direct process. The bitlist, e.g., is directly stored in a byte array representation now. The original storage is a serialised version which must be extracted from a byte array representation before using it. This indicates that (at least) one additional step must be taken to create / manipulate / read the bitlist. The H2 database already has direct access methods for byte arrays. Thus, representing the bitlist inside a byte array is a very direct way of storing data. The remaining data, such as the secondary inverted index root node or the term id, are also saved as native data types like long or integer values. Thus, there is no need of (de-)serialisation before and after the use of these data types. Based on the values given in the figures, the effect of this is tremendous. Each of the storage structures is changed in this way. Hence, the initial inverted index now has one entry for a string value and, potentially, a term id in an integer representation. The R-Tree stores a (manually) serialised version of the spatial element, the bitlist as byte array and the secondary inverted index root reference as long value. The secondary inverted index only saves an integer value as the term id. All of these values are now native data types which are also supported to be stored directly in database pages. Therefore, no (de-)serialisation is required to manipulate them, which means that at least one step which tends to be time consuming can be skipped except for the spatial component. Yet, they may still be stored temporarily inside a LRU cache as described in the first iteration introducing caching mechanisms. This also means that only the spatial components are used inside the cache and no other value types are processed there. Hence, the caching may focus on these parts, which also tends to make it more efficient as it can concentrate on one part of the data types.

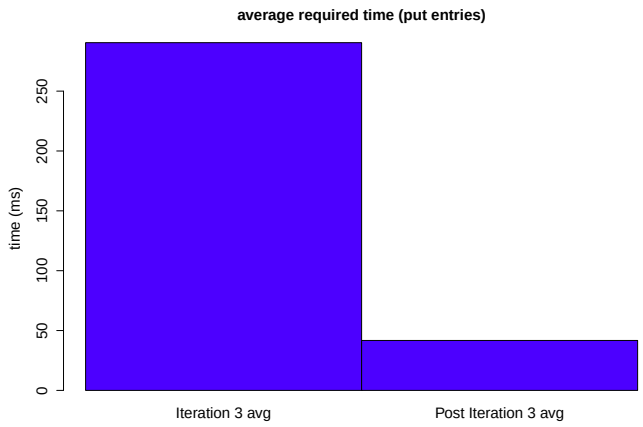


Figure 5.67: Put To Secondary Inverted Index Function Comparison

Figure 5.67 displays the improvement factor of the `putEntries` phase which is the primary phase selected for optimisation in this iteration. Based on the outcome of the empirical study executed on the Wikipedia corpus, it was decided to implement the HYBRID strategy for the manipulations of the secondary inverted index. As the secondary and initial inverted index share the same algorithms for manipulation, a potential improvement in the secondary inverted index also refers to the `rootBtree` phase. It cannot be stated definitely which optimisation step in this iteration the effects seen in figure 5.67 result from because also the internal data management have changed. Obviously, the resulting runtime is only $\approx 12.71\%$ of the initial runtime which leads to a speedup of $\approx 87.29\%$. Unfortunately, due to the two optimisation steps performed in this iteration, it cannot be clarified unambiguously which of them is responsible for this enormous effect.

Function	Duration Pre	Duration Post	Amount
rootBtreeInsert	250, 287.539	15, 808.135	39

Table 5.19: Pre and Post Values for Root B-Tree Manipulation

The exact values for function execution before (pre) and after (post) the optimisation for the initial inverted index can be depicted from table 5.19. The amount of calls obviously stays the same as the configuration did not change in the pre- and post-test phase. The results for the initial inverted index are even more considerable. The execution of the post-tests regarding the initial inverted manipulation only requires $\approx 6.32\%$ compared to the pre-test. That indicates an improvement of 93.68% for this phase. The increased gain in performance compared to `putEntries` results from the usage differences of these two functionalities. First, the term entries are inserted to the initial inverted index. Hence, this phase adds as many elements into the initial inverted index as stored in one document entry performing its work, e.g. extracting "overflow" entries. Thus, due to the one time execution of this phase, caching effects on database pages are more likely to be applied there than in the `putEntries` phase. The secondary inverted index manipulation takes place at each leaf element of a leaf node affected by the distribution algorithm. Therefore, this phase is carried out multiple times at multiple entries of the R-Tree which leads to the fact that the caching of pages cannot be as efficient as in the initial inverted index manipulation. This manipulation is executed on exactly one inverted index whereas the process of putting elements to the secondary inverted index runs on multiple inverted index structures. Besides, the initial inverted index also stores all term entries, which means that it tends to store more entries than the particular secondary structures. Hence, also the inverted index optimizations work better for the initial than for the secondary inverted index structures. Nevertheless, the entire optimisation approaches introduced a massive performance gain in each phase of the hybrid index.

5.2.4.5 Summary

With the primary focus of this iteration being the improvement of inverted index structures, two options are selected. On the one hand, the optimisation and application of general inverted index manipulation algorithms are introduced based on literature and empirical studies. On the other, a total restructuring of internal storage mechanisms is implemented. Both adoptions show tremendous effects, although, in some cases, it cannot be pointed out unambiguously which of the changes is more influential regarding the values in the evaluation. Especially as to the inverted index structures utilised respectively, it cannot be construed which of the modifications in the internal algorithms cause the particular improvements. The effects may result from the improved storage behaviour of the inverted index, because a lot of work in this iteration is performed to construct an appropriate strategy used to decide the proper place to store a tuple to. This adoption of the strategy is very likely to have positive effects on the overall reorganisation performance. As shown in the analysis of the study in this iteration, it is obvious that the alteration of the strategy may result in an improved behaviour of insertion as well as retrieval procedures. Thus, the adoption of the strategy is supposed to have a large effect on the reorganisation time as it is executed very frequently when tuples are placed to their final destination, the secondary inverted index. Yet, the switch in the basic storage layout also affects the entire reorganisation procedure. It could be shown that the change in the storage layout introduces a more direct and thus more efficient way of accessing internal tuple data. As this change also affects the time required to modify tuples in the (secondary) inverted index structures, the exact influence of the respective changes may not be directly quantified. However, it may be stated that at least the combination of both optimisation approaches turns out to be beneficial for the entire reorganisation procedure.
Due to the internal changes of the storage mechanism the search algorithm had to be modified as well. Yet, this only relates to the extraction of particular values from the rows. Thus, all parts of the search algorithm had to be modified. Besides these alterations, the retrieval inside the respectively used inverted index had to be changed, too. These changes were introduced based on the adoptions in the manipulation algorithm. The retrieval must react on the hybrid storage strategy which either stores items consecutively or in an additional inverted page. Thus, the respective parts have been changed, too, in order to be able to retrieve data properly again.
With reference to the search efficiency, it can be stated that this part has not worsened at all. The main changes introduced to the algorithms in the search part are based on the inclusion of the HYBRID storage option of inverted index management. As displayed in the empirical study, the worst option is the LEAF method which is used in advance of this iteration. It is employed by the inverted index as long as there are less entries than would fit into one page. Consequently, in the worst case, the search effort after this optimisation equals the behaviour of the LEAF strategy. As it is applied before this optimisation, the effort is at maximum as high as in the LEAF strategy case. For this reason, the search effort is at least the same as in advance. The empirical study shows that on average the strategies EXTRA and HYBRID outperform the LEAF strategy. Therefore, also positive side effects on the search effort are expected because of the adoptions performed by this iteration.
The effective changes deployed to the algorithms in this iteration are displayed in figure 5.68. The adoptions applied in the comparison and storage structure modifications refer to nearly each part of the algorithms. They are utilised in each case the placement or loading of a tuple from a database page is required. As each of the storage structures is affected by this alteration, marks are left out from this picture.
The alteration of the inverted index treatment mainly affects two phases. On the one hand, the initial inverted index is optimised in this step. Yet, this is not the main target of the adoptions. In addition, based on the fact that the number of entries directly referred to by the initial inverted index is quite small, the changes do not have a big effect, here. On the other, with both inverted index structures sharing the same code base, the secondary inverted index is modified too, which is the main target of this optimisation step.

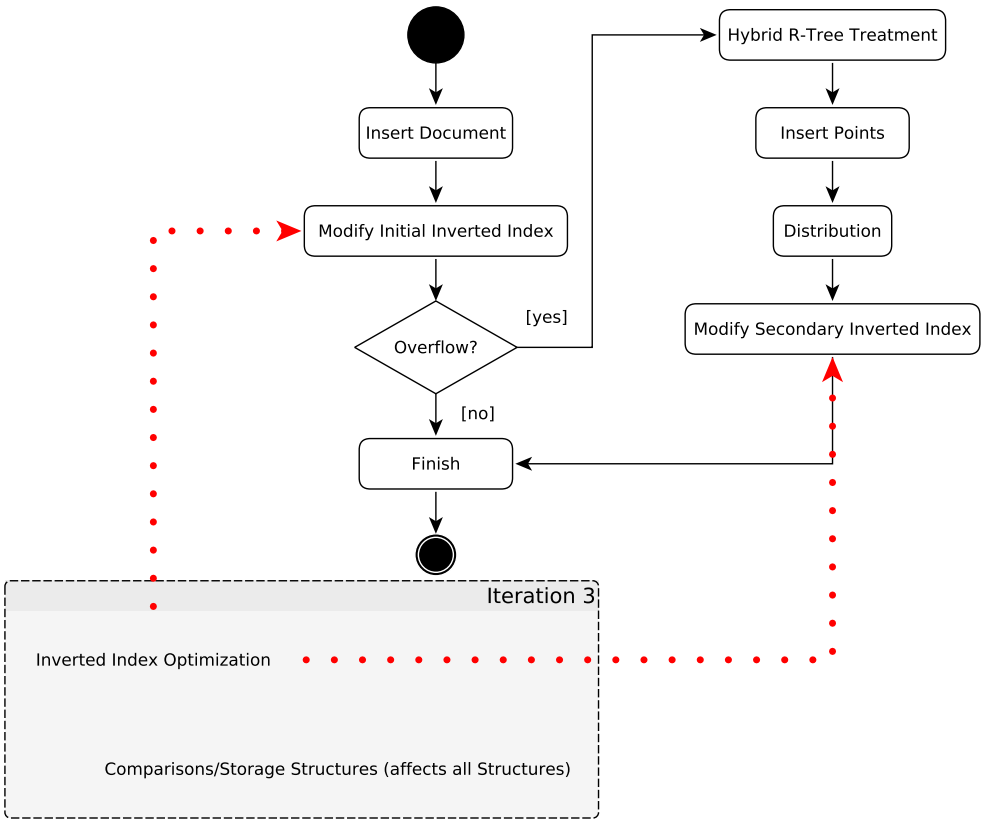


Figure 5.68: Activity Diagram of Changes Performed in Iteration 3

5.2.5 Pre-calculation of Item Insertions and Advanced Inverted Index

As already discussed in advance at multiple stages of the optimizations, caching and pre-calculation are an important part of the improvement of database related operations. Yet, the memory consumption of these operations must always be kept in mind. A larger amount of main memory may be allocated. Still, this quantity must be freed at pre-defined points of the execution of algorithms. Besides, the memory may not be kept permanently. Therefore, buffers mostly consist of size limited queues. In comparison to caching values, the pre-calculation is another option. It takes a pre-defined set of elements and tries to generate a distribution of them which can be utilised later on. It is always a good idea to observe on the required quantity of temporary memory here, as well. This section analyses the behaviour of the tree distribution functionality and optimises it by pre-calculating the assignments of point values to term ids pointing to the documents. Therefore, these assignments, which are needed at the very final stage of inserting the intersections of point to term ids at the secondary inverted index structure, are already generated at the beginning of the hybrid tree modification procedure. Besides these pre-calculations, the B-Tree manipulation (insertions or deletions) may also be optimised as there are still more advanced reorganisation techniques available for this issue. Some well-known techniques and improvements are applied here, as well. It must be noted that the procedures described now are basically derived from two optimisation iterations and combined in the description in this section.
Therefore, this section is also decomposed in multiple measurement schemes. The measured data result from a study executed over all iterations in order to compare them finally. This study is described in a little more detail in chapter 6. Basically, it is used to depict the results from the entire optimisation process. Hence, the data used as an input set here comprise a little smaller set of documents. However, it is hard to compare the data of multiple iterations if the amount of documents inserted into the database table varies. This is basically due to the fact that effects

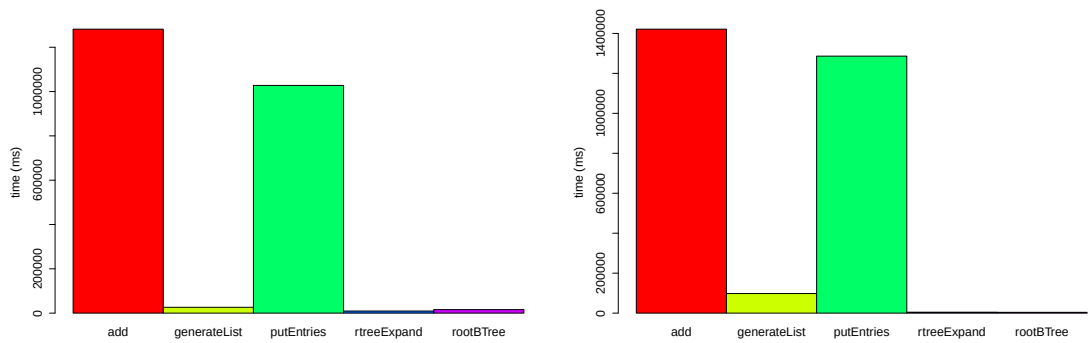
inside the program tend to diversify on different input data. Thus, there is no possibility of comparing an iteration using 500 documents as input dataset and another one using 1,000, where the 1,000 might also include the 500. However, the effects arising in the larger input set might not be directly comparable to those from the smaller one because hypothetically these effects could result simply from the bigger number of documents in the bigger sets. The study specified in section 6.4 was executed in order to prevent these effects by applying a pre-defined set of documents. Based on the fact that the data of more than one iteration are utilised for this description, it is necessary to find a representative way of comparing multiple resulting data. Therefore, the data from the study are taken into account for the description of the optimizations now.
The settings for the two inspected iterations are basically the same as before. However, the amount of elements is smaller in this case. Only 38 repetitions of the test suite are executed for measurements reasons. This is mainly imposed by the inefficiency of the initial implementation. The amount of documents used in the inspected iterations is thus $n = \lceil \frac{38}{2} \rceil \cdot 50 + \lfloor \frac{38}{2} \rfloor = 969$.

Parameter	Value	Explanation
HLimit	200	Artificial Upper Bound
Element Count	5	Amount of Elements per R-Tree node
Document Count	969	Amount of Documents to be inserted
Dataset	Wikipedia	Dataset to be processed (Wikipedia sorted by ids)
Split	R*-Tree Split	R-Tree splitting method
Choose Subtree	R*-Tree Choose Subtree	Method for Selecting a proper subtree to place an element in

Table 5.20: Settings for the Pre-tests Test Suite Run

The detailed list of settings applied to both test executions valid for the given results is displayed in table 5.20. Detailed test data for both iterations are also present, internally, which, however, cannot be compared directly. Iteration 4 is carried out with 39 measurement points (78 repetitions), iteration 5 with 76 (152 repetitions). The data are also present and may additionally be inspected for selected features.

5.2.5.1 Basic Observations



(a) Phases Pre Iteration 4

(b) Phases Pre Iteration 5

Figure 5.69: Phases of Iteration 4 and 5

Figures 5.69a and 5.69b show the numbers for the measurements of iterations 4 and 5 which are analysed in this section. It can be seen that the dominant phases regarding runtime are the

phases of placing items to the secondary inverted index. The phases `rootBtree` as well as `rtreeExpand` do not play an important role here.

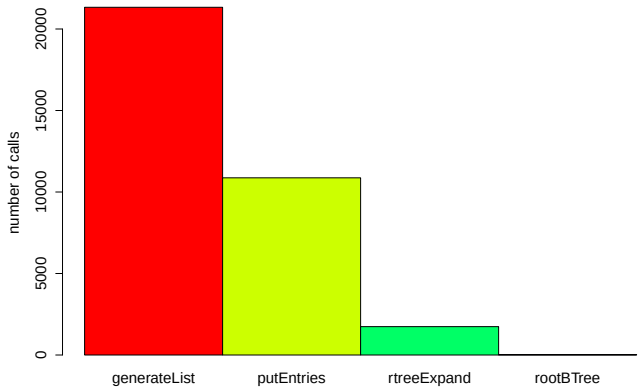


Figure 5.70: Number of References to Phases in Iteration 4 / Iteration 5 Pre-Tests

The amount of executions of the particular phases can be depicted from figure 5.70. It shows that the list generation is called most, followed by the secondary inverted index manipulation phase. The initial inverted index execution does not contribute much, neither to the runtime nor to the number of executions. The list generation phase seems to be very efficient before of the respective phases. Thus, iteration 4 focuses on the optimisation of the secondary inverted index structures. An individual analysis using the paths analyser tool can be omitted here because the optimisation of the secondary inverted index is obviously aimed at. Therefore, general (or more specific) optimisation approaches need to be investigated for the application in the inverted index structures.

Function	Duration	Amount
putAndAdjust	797, 459.33	62, 705
adjustTree	627, 617.623	75, 093
adjustAnchoredPageIds	604, 325.01	56, 843

Table 5.21: B-Tree Related Functions

The exact values for B-Tree dependencies can be found in chart 5.21. This table lists three functions depending on each other. `putAndAdjust` executes `adjustTree` calling `adjustAnchoredPageIds`. The latter is responsible for notifying associated pages like inverted pages about a change in the position of the given page. This might be a crucial task as, internally, no decision can be taken at first glance, if an inverted page is present for the certain page or whether it just references a document stored inside the user table. Hence, potentially, this function must scan forward to the first item (or it is the first and only one) whilst loading additional pages inside the B-Tree. As this procedure is executed on each level for each insertion, it takes a long time probably without performing a lot of output. Therefore, this is a potential candidate of optimisation. The second method which badly contributes to the runtime is the choice of a subtree and also the process of finding a place inside a page. Currently, this is implemented as a linear scan through the pages inside the inverted index structures. If many pages must be searched for a proper place and the item to be found is near the end of the array of items, this will negatively contribute to the runtime.
The second target of the two phases is to optimise the list generation. Nevertheless, the manipulation of the secondary inverted index structures is still dominant regarding the numbers given in figure 5.69b. The selection of this phase is explained in the following subsection. At first sight, it is not obvious why the list generation and the algorithms related to it are taken into account for the optimisation. However, a comprehensible explanation is given in subsection 5.2.5.2. In fact, the analysis of the individual phases results in a recommendation of the `putEntries` phase in both inspected iterations. Nevertheless, sometimes a detour to other functionalities and the reflection of a higher level process can also be successful.

5.2.5.2 Solution Alternatives

The solution alternatives listed here refer to both inspected parts. First, the alternatives to lower the demands of the B-Tree implementation and then the distribution scheme are listed.

5.2.5.2.1 B-Tree / Inverted Index Actually, the solution of the problems related to the B-Tree cannot be found in literature as this structure is customised in order to fulfill the needs of the hybrid index structures. Hence, a deeper investigation of the algorithm must be carried out in order to optimise it.
The current situation results from a previous optimisation run. The restructuring of the inverted index leads to the fact that either an entry is found in an inverted page or directly inside the B-Tree leaf. Possibilities had to be created to distinguish between the direct LEAF storage or the EXTRA storage in case of the HYBRID approach. In the initial implementation, the elements are stored one after another sorted by the document heap reference to be able to retrieve items by using the NRA approach, again. Therefore, it is clear that each individual entry inside the inverted index directly refers to one document. The introduction of the HYBRID storage of inverted index elements, however, introduces additional differentiations. A dummy element is created referring to a NULL element which is only present if the elements are stored in the LEAF strategy. This dummy element then symbolises this methodology. If it is absent, the references are stored in an external inverted page utilising the EXTRA strategy. If a node inside the B-Tree is split, it is necessary to signalise the respectively referenced pages it was moved to. Thus, a procedure is executed to update the particular references. This method searches for previous entries equalling the currently inspected one to determine whether it follows the LEAF or the EXTRA strategy. Currently, this method follows all references to the particularly equal items to the front of the list where the dummy element should be located. If it is present, the existence of the LEAF strategy is indicated. Otherwise, additional inverted pages are indexed here.
This entire procedure is executed every time an item is added to the secondary inverted index structures. Especially the scan for previous items until the dummy is found or not is an expensive operation. This potentially results from the fact that it is necessary to load, at most, one additional page if the dummy element is stored in the node preceding the currently inspected one. This is executed if its first element is the same as the B-Tree entry of investigation in order to check for the dummy, here.
Based on the considerations of the strategies, it is obvious that only multiple items of one kind are persisted inside the B-Tree if the LEAF strategy is applied. If EXTRA is utilised the respective leaf consists of exactly one item pointing to the inverted pages. Hence, there is no direct need for searching the very first item of the list. It is sufficient to find one additional item stored inside the B-Tree. Thus, the solution alternative in this case is to scan for an additional item. If at least one additional item is present, LEAF is obviously the strategy applied. For this reason, exactly one additional item (either previous or next) is checked for presence. That means that the need to load a supplementary previous page may be skipped if one entry inside the B-Tree can be found which equals the currently inspected one. If the inspected item is stored at position zero, first a check is performed to detect if there are any additional equal items following. Otherwise, as a last step, the previous page is loaded and examined. This avoids, on the one hand, a large number of comparisons as they are reduced to exactly one and, on the other, also the requirement of having to load an additional page (the previous one) if the requested item is not stored as first element in the given B-Tree node. Furthermore, these checks must only be performed at leaf level because nothing but the situation of linking to inverted pages may occur if the EXTRA strategy is applied. This is a rather small optimisation which could potentially lead to a huge outcome because in some cases the additional loading of a previous page may be skipped now.
The current search process inside the inverted index operates on a sorted list of entries. They are either ordered lexicographically in the initial inverted index or by the numeric value for the secondary inverted index. When trying to find a proper place an item is located in or must be placed at, the entire list needs to be checked. In the initial implementation, it is necessary to perform the check for each item in the inspected node. If the respective element to be handled

resides at the very end of the list, the entire set must be compared with the item to be found or placed. This results in a linear search complexity ($\mathcal{O}(n)$). These checks need to be executed in each node of the B-Tree until a leaf node is found and the respective position can be determined in there also with a linear complexity. Although the B-Tree guarantees a logarithmic search effort, the internal searches potentially require a large amount of time. The linear complexity given in internal node searches are not very desirable in this case. A solution for this issue is, e.g., to apply a binary search in internal operations of finding places (see, e.g. [61, p. 409 ff.]).

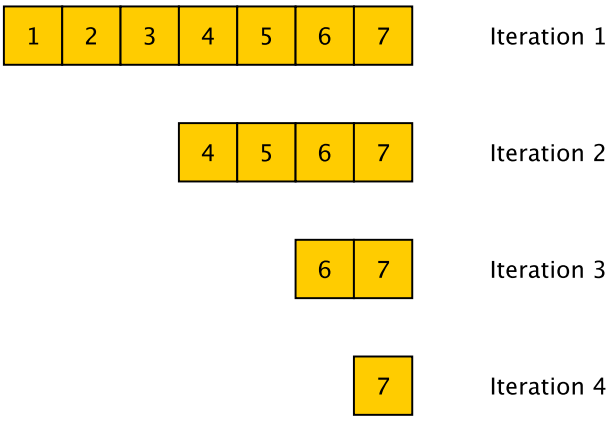
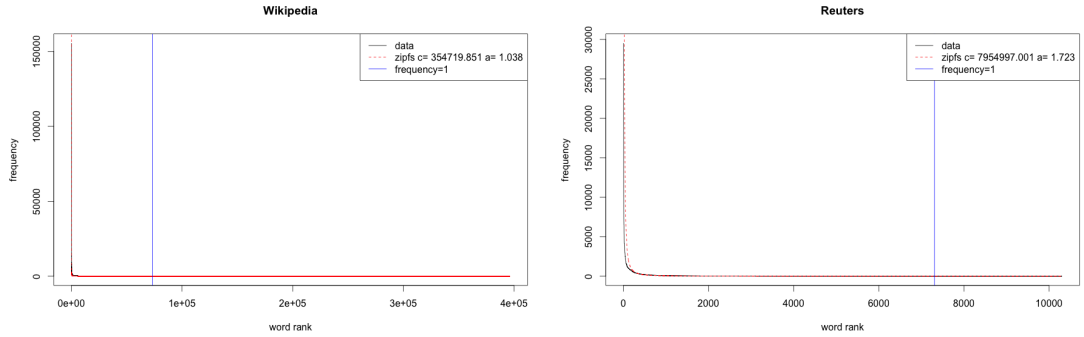


Figure 5.71: Concept of a Binary Search

A graphical conceptualisation of a binary search is displayed in figure 5.71. A search is issued for item 7. This is actually the worst case for a linear search because seven comparisons must be carried out in order to find the item finally. The binary search using a bisection based approach only requires four iteration steps to find the respective item. The binary search always takes the "middle" element separating the list of sorted entries in two halves recursively until the item is found. At maximum $\lceil \log_2 n \rceil$ comparisons are necessary to detect one item compared to n comparisons of a linear search. Thus, the amount of comparisons to find internal elements can be lowered by using this approach. This is an important feature because at least inside the secondary inverted index structures, a lot of items are placed for which a proper place must be found. Thus, numerous comparisons are carried out to position the items.
A further subject of investigation is the distribution of points. The basic cause for this consideration is the extensive manipulation of secondary inverted index structures. It is most probably worthwhile investigating the secondary inverted index structures. For this intent, a look is taken at the frequency of the respective points inside the documents.



(a) Zipf's Law Distribution of Wikipedia Points

(b) Zipf's Law Distribution of Reuters Points

Figure 5.72: Zipf's Law Distributions of Wikipedia and Reuters Regarding Points

The Zipf's Law distribution of Reuters and Wikipedia points can be seen in figures 5.72a and 5.72b. The red dotted lines display the values of Zipf's Law calculated by a regression analysis.

These are basically the same figures as in subsections 5.1.3.2 and 5.1.3.3. A blue line is added for this illustration. It separates the point values occurring only once from these occurring more frequently. With respect to the Reuters collection, the relation of points with frequency one to the total amount of point values inside the Reuters collection is $\approx 29.08\%$. The Wikipedia collection has a much larger quantity of elements occurring only in one document of the entire corpus. The value here lies at $\approx 81.47\%$. That is to say that a large portion of the entire point set is only referenced by exactly one document. For Reuters this value is $\approx 70.92\%$ and for Wikipedia the number of points occurring more than once is $\approx 18.53\%$. The more interesting corpus for this analysis is the Wikipedia corpus as there is a large number of points with frequency one.
The hybrid index is set up based on the distinction of often and seldom used terms. Frequently used terms are likely to occur in all documents inside the collection. That means that (nearly) all points which are used inside the collection also occur inside the R-Tree part mixing the non-normalised with the normalised part. Therefore, the distribution of points and references inside the R-Tree is similar to the ones shown in figure 5.72. That indicates that also in the leaf nodes of the hybrid R-Tree, a large number of points exists which is only present in exactly one document. Nevertheless, multiple terms to be stored in a secondary inverted index structure might be valid for one normalised point value. Each of the R-Tree elements (also at leaf nodes) store a bitlist to indicate the presence or absence of the respectively referenced terms. Thus, already in this stage, the information about a certain term being stored below this element is given. So, if only one document is pointed to by a certain point, the intersection to be generated between points and terms can directly be derived. For this reason, there is no need to set up an external structure using an inverted index accessed by a B-Tree to store information about this intersection as always exactly the same document is referred to, here. That indicates that a specialised storage procedure can be used which omits the use of a B-Tree to perform the intersection. A possible solution is the storage of a flag indicating that no secondary inverted index needs to be loaded in conjunction with the real reference to this particular entry. Hence, for 70.92% of the points from the Wikipedia corpus and 29.08% from the Reuters collection, the modification of a secondary inverted index structure can be avoided by this procedure which is supposed to speed up the manipulation operations.
Another optimisation approach which could be applied is the introduction of bulk operations on the inverted index. Previously, for each item representing a term id to a document reference, one insertion operation is executed on the secondary inverted index. However, at least for terms with the same ids, bulk operations can be utilised taking the list of elements directly to the leaf nodes where the decision is then taken whether to insert them into an inverted page or directly at the leaf node. In some cases, the memory restrictions for the item then move it from the leaf storage to an additional inverted page in order to fulfill the HYBRID storage strategy of inverted index structures here. Thus, it is probably meaningful to treat all the respective elements at once in order to reduce the number of individual insertion and reorganisation operations in the secondary inverted index structures. Iteration 5 introduces additional optimizations based on this approach where items fitting in one node are put all together instead of having to descend exactly the same path multiple times.
These are the main optimisation possibilities for the inverted index obvious due to measurements and basic observations.

5.2.5.2.2 Distribution As already mentioned, the distribution of items including the generation of lists of elements valid for the particular subtrees is an important operation. The actual phase was subject of investigation of preceding iterations. Unfortunately, this phase is still quite inefficient in iterations 4 and 5. This cannot directly be depicted from the figures at hand in the basic observations. It is, however, given in the actual numbers of the iteration measurements which are omitted here owing to the lack of comparability.
Two options are conceivable to solve issues while generating the list. As already mentioned, either pre-calculation or caching may be a solution to improve the performance of specific parts of a software. A caching mechanism could be introduced as a first approach for being able to generate the sublists more efficiently. The current implementation gets a list of term ids each of which points to a list of documents as parameter. The points stored inside the document lists

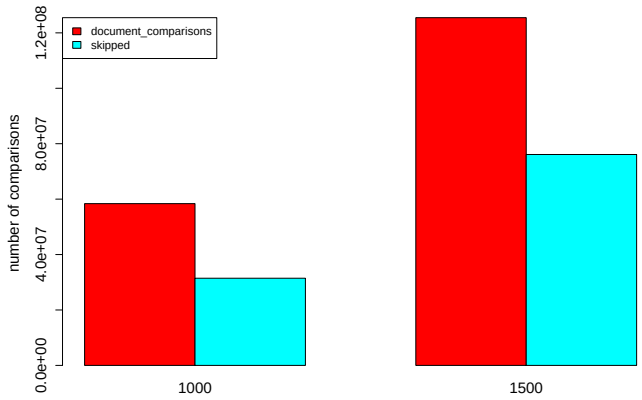
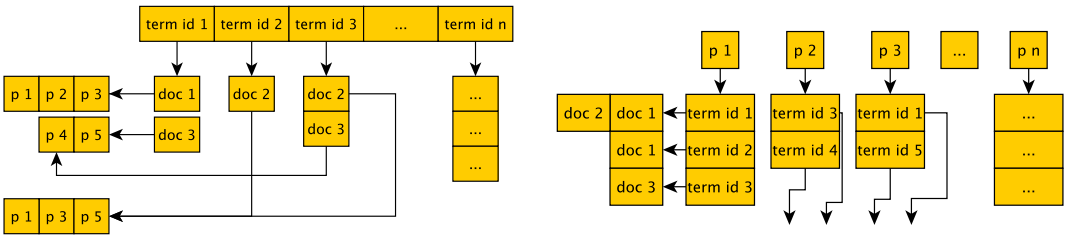


Figure 5.73: Comparisons and Skipped Comparisons of Documents

must be checked to see whether they are either inside the region represented by the spatial component of an R-Tree element in an inner node or if they are equal to a given point in a leaf node. The respective documents are, however, potentially stored in the lists multiple times. It seems to be inefficient to check one document multiple times for one element inside the R-Tree. Figure 5.73 shows the amount of comparisons in relation to skipped comparisons. A caching mechanism which stores already checked document ids in a set representation is built to compare these values. This is an experimental validation for 1000 and 1500 inserted documents. The Wikipedia dataset is used for this validation. It can be seen that by the use of comparison result caching only 53.89% (1000) or 60.65% (1500) need still be executed and the remaining ones may be skipped. This is a gainful property because the amount of comparisons rises a lot. Still, this process seems to be inefficient because the lists are separated to the particular spatial elements where they fit in and, finally, a list of elements needs to be built valid for the point given to insert them at the final destination.
Hence, in order to avoid the iteration through each list of documents and each point in these lists for each inspected R-Tree element, a more direct way seems to be imaginable. Besides, it is potentially possible to pre-generate specific portions to be distributed. Thus, a feasible solution in this case is to create mappings holding the assignments that will be created at leaf level already at a very early stage.



(a) Term List Object as Passed Through the Reorganisation

(b) Desired Output for Insertion

Figure 5.74: Term List and Assignment

After the insertion operation on the initial inverted index, the lists of overflowing items are present. These lists are directly obtained from the initial inverted index. However, it is already possible at this stage to generate an assignment mapping. The original state of the overflow list is displayed in figure 5.74a. The term ids point to documents which refer to a list of points themselves. This is the original list from the initial inverted index which is passed through the tree during the reorganisation procedure. A large amount of comparisons needs to be executed

in order to generate sublists valid for the respective subtrees. Nevertheless, at the secondary inverted index structures, an assignment is required which refers from a point to a list of term ids pointing to documents. Thus, an inversion of this list is required in prior to insert the elements to the secondary inverted index structures. Besides this fact, a large amount of iterative list parsings must be executed during the distribution of items. A more direct way seems to be possible if an assignment is pre-calculated which has a point value as its key referring to the term ids that point to the documents where this particular combination occurs in. An example for such a distribution is given in figure 5.74b. This results in the possibilities to check the points directly and does not require generating sublists of the structure given in illustration 5.74a. Thus, having pre-calculated an associative array the point values may directly be passed to the final destinations. When a point is reached, the list of elements the point refers to can be directly taken and put to the secondary inverted index structures. On the one hand, this procedure avoids a large amount of comparisons and, on the other, a large amount of sublist generations. Restructuring this procedure seems promising for optimising the list generations as checks for documents need not be performed any more and the points may directly be checked. However, an increased effort must be taken to pre-generate the mappings. The basic intention of this suggestion is that the pre-generation can be performed quickly and enables faster comparisons during the traversal of the hybrid R-Tree. Thus, the summarised time of pre-generation and distribution is lower than the time for the individual comparisons. It is probably obvious that the comparisons will tend to be faster because they must only be executed once per point value at each element. In contrary, it is necessary to perform the comparisons of point values and documents multiple times using the initial state of the assignments. Therefore, there is the basic hope that the new approach is faster.
The current procedure for distribution also seems to be a bit inefficient if changes are applied due to the allocations. On principal, the mapping which could be used directly points from the point values to the pre-generated lists of entries to be placed at secondary inverted index structures. Because of the pre-generated lists of elements derived as assignments at the beginning of the distribution phase, it is now imaginable to place the items in a more direct way. The current implementation traverses the tree to descend to a leaf node where all items satisfying equality of the point value are put to the secondary inverted index structures. As already pre-calculations are performed pointing from a point value to the list of elements to be put to the secondary inverted index there, it is also possible to place the lists straight at the leaf node. Thus, the spatial component can be found by using the `findLeaf` functionality of the R-Tree which subsequently delivers candidate elements for a particular leaf node. All inverted index manipulations may be done at this node and the tree can be reorganised later on.

5.2.5.3 Selected Solutions

The solution alternatives for the secondary inverted index as well as hybrid R-Tree operations are listed in the previous subsection. Generally, no real alternatives are listed in this subsection. The basic observations showing an enormous demand of time during the secondary inverted index manipulation operations are enumerated and the solution alternatives can be seen.
Nevertheless, all of these alternatives are introduced for the given two iterations.
The distribution scheme is changed twice, once per iteration. The first approach is the attempt to cache the execution of comparisons. There, each document which is compared to a respective R-Tree element is saved in a cache using a document identifier in conjunction with a boolean flag storing the comparison result. Based on very fundamental tests, the effect of this caching can be shown. However, this does not seem to be an optimal solution at this stage. Hence, the pre-calculation of tuples is also set up by the use of an associative array (`java.util.HashMap`).
The solutions selected for this iteration are:

1. Inverted index scanning of previous items,
2. bisection based search in B-Tree elements,

3. special storage of secondary inverted index structures of point values occurring in only one single document,
4. bulk operations inside the secondary inverted index structures,
5. pre-generation of assignments for leaf elements of the hybrid R-Tree to insert them to the secondary inverted index structures more directly,
6. more direct way of distributing items via `findLeaf` and subsequent placement.

This displays that the changes applied here are merely internal changes restructuring the algorithms to optimise the runtime.

5.2.5.4 Evaluation

The evaluation is executed in two steps whereas only the total overview is analysed, here. This section discusses the results of two individual iterations because only internal changes are performed. Based on the data resulting from a big test case executed with the same number of elements as overview, the settings stay exactly the same as given in the pre-test section.

Parameter	Value	Explanation
HLimit	200	Artificial Upper Bound
Element Count	5	Amount of Elements per R-Tree node
Document Count	969	Amount of Documents to be inserted
Dataset	Wikipedia	Dataset to be processed (Wikipedia sorted by ids)
Split	R*-Tree Split	R-Tree splitting method
Choose Subtree	R*-Tree Choose Subtree	Method for Selecting a proper subtree to place an element in

Table 5.22: Settings for the Post-Tests Test Suite Run

The parameter setup of the post-tests can be depicted from table 5.22. Due to the fact that the data taken for analysis are generated by a test case inspecting the entire optimisation runs, the data stay identically equal to those in the pre-tests. Besides, no external parameters have changed during the iterations as outcome of an optimisation.

5.2.5.4.1 General Evaluation The entire process is displayed in figure 5.75. Iteration 3 gives the time average required time per item in the pre-test phase which is the outcome of iteration 3 and iteration 5 is the post-test of iteration 5. It can easily be seen that the time needed is cut off by $\approx 78.18\%$ in total. Thus, the average time necessary for the insertion of one element is lowered to 21.82% of the original one. It must be noted that two phases drop out as the R-Tree modification and list generation are carried out together after this iteration. Hence, no exact numbers can be stated here.
The secondary inverted index and thus the phase `putEntries` is subject of investigation in this optimisation iteration. The results for the `putEntries` phase as well as the initial inverted index manipulation are shown in figure 5.76. The initial inverted index also benefits from the optimizations. Mainly two features play an important role here. The scan for previously stored items as well as the bisection search leads to a decreased amount of comparisons and also helps by avoiding to load additional pages. In average, the manipulation of the secondary inverted index structures only takes $\approx 22.87\%$ of the original time. As this phase is one of the

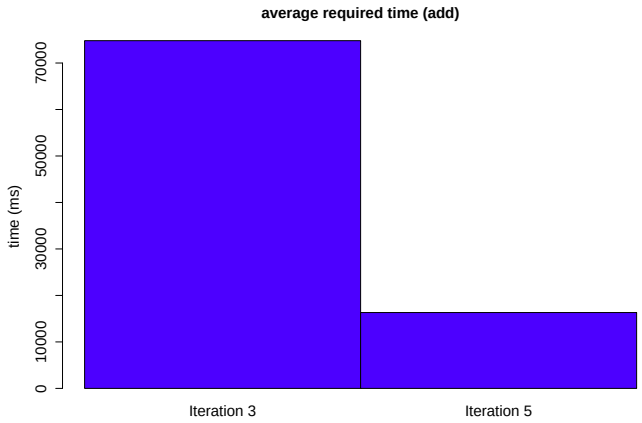


Figure 5.75: Add Function Comparison

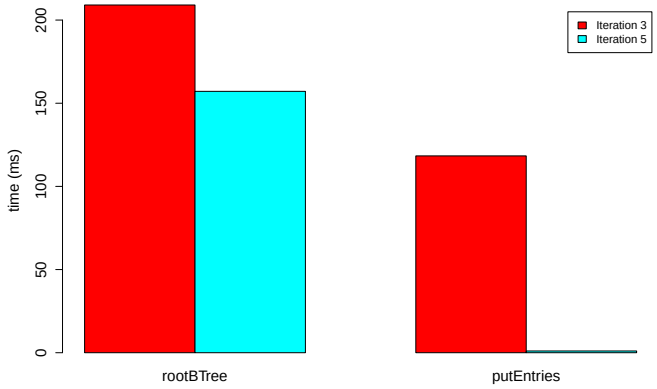


Figure 5.76: Inverted Index Function Comparison

main targets of the present iteration, it may be stated that the optimisation effects are as high as desired at this stage. The detailed comparison of methods is carried out later in this section. Distribution refers to all actions taken while traversing the hybrid R-Tree including the deployment of items inside the particular subtrees as well as the secondary inverted index manipulation. Figure 5.77 shows the values for pre- and post-tests of iteration 4 and iteration 5. It is obvious that this phase could be significantly improved. The details at this stage cannot be retrieved from this figure because a lot of internal logic also changes, which is evaluated in the individual analysis.

5.2.5.4.2 Individual Evaluation The individual evaluation for this section is split in the two respective parts, inverted index and distribution of items comparisons.

Inverted Index Regarding the secondary inverted index, three major changes are applied in this iteration. The effects of the scanning is not directly noticeable from the test data. The bisection based searches may be mainly found in the amount of comparisons carried out internally. This could also result from bulk operations because with this approach a different quantity of comparison operations must be executed, too. A comparison of the calls to the insertion of B-Tree items is shown in chart 5.23. Two different comparisons lead to the extremely

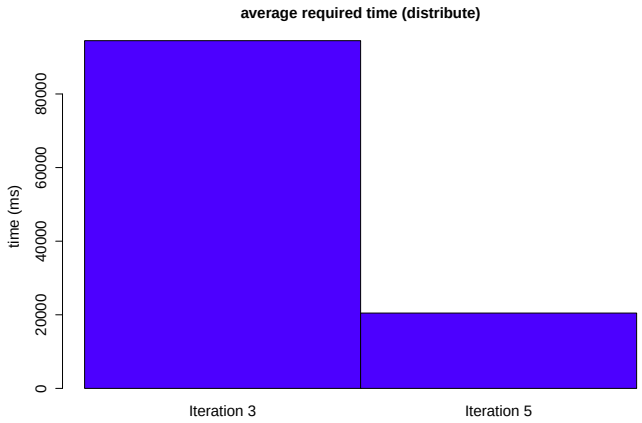


Figure 5.77: Comparison of Distribution Function

Function	Duration	Amount
putAndAdjust (pre)	797, 459.33	62, 705
putToBtree (post)	5, 839.232	20, 116

Table 5.23: Pre- and Post-Test Comparison of Placing Items inside Secondary Inverted Index Structures

lowered amount of executions. This procedure is not executed at all if a point is only contained in one individual document which is placed to the secondary inverted index. As already shown in the solution alternatives, this occurs frequently in both inspected corpora. Still, the amount is $\approx \frac{1}{3}$ of the original one. The cumulative runtime is only $\approx 0.73\%$ of the initial one instead.

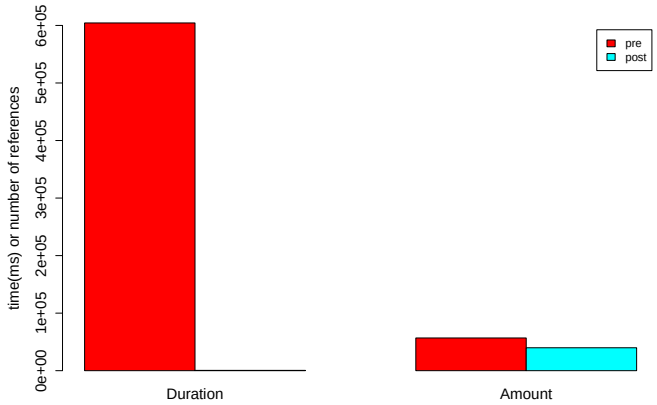


Figure 5.78: Adjustment of Linked Pages

The main cause for this enormous improvement is displayed in figure 5.78. It shows the amount of calls and the total runtime of a function called `adjustAnchoredPageIds` which is responsible for getting the references right again. This is the place where the optimisation of the process of scanning the previous items comes into account. In most cases for the new approach no extra page needs to be loaded. Loading the previous pages in the original version of this algorithm is the main time consumer in the version present before this iteration.
Table 5.24 shows the values for row comparisons before and after the respective iterations. The average duration of one comparison has not changed much. However, the total amount of comparisons has been lowered essentially. Hence, the total time required for comparisons

Function	Duration	Amount
Before	2,723.369	16,264,147
After	71.967	582,683

Table 5.24: Row Comparisons Before and After Optimisation

decreases by $\approx 97.36\%$. Two adoptions lead to this dramatically lowered quantity. On the one hand, the direct storage of points occurring only in one document is responsible here. On the other, also the introduction of an improved search variant using the binary searches in internal B-Tree nodes contributes to this number as well. It is not directly recognisable which of the changes leads to this improvement.
These are the most notable changes in runtimes of the inverted index manipulation functionalities. As already seen in the general evaluation, the changes also have an influence on an improved runtime behaviour of the initial inverted index.

Distribution The evaluation of the distribution must be executed on two different parts. The process of distributing items through the hybrid R-Tree is split in the generation of the assignment mapping and the actual R-Tree manipulation. This entire phase needs to be compared to the two steps of R-Tree expansion and distribution which is the initial state.

Function	Duration	Amount
generateList	24,623.55	21,332
putEntries	1,356,351.351	10,870
rtreeExpand	5,007.947	1,741

(a) Pre-Optimisation Values

Function	Duration	Amount
genAssignment	153.501	15
rtreeExpand	306,849.208	15

(b) Post-Optimisation Values

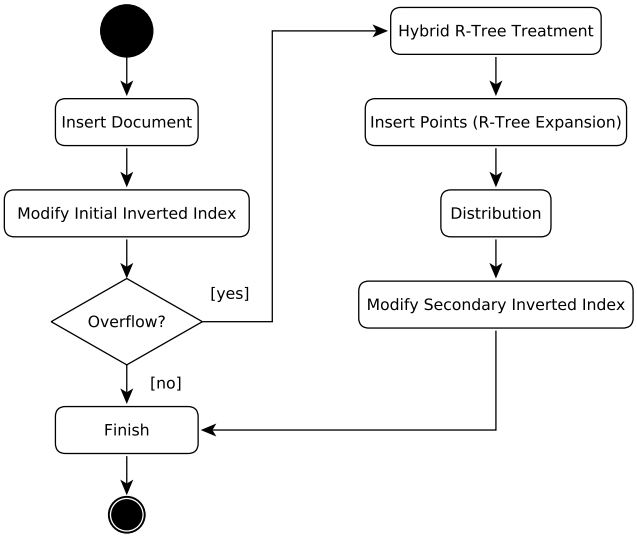
Table 5.25: Distribution Times Before and After Optimisation

The exact effects of the optimisation regarding the distribution functionality are displayed in table 5.25. The numbers show that the changes in the algorithms have enormous effects. Additionally, the generation of the mapping assignment does not contribute much to the total runtime. Thus, the pre-calculation of the lists can be performed efficiently in main memory and the actual distribution of the items through the tree is also much faster. The `rtreeExpand` phase is slower based on the figures. In the new version, however, this phase also executes besides the insertion of entries to the R-Tree also the distribution which is separated in the version before. The actual time differences are already displayed in figure 5.77, which also shows a tremendous improvement of the entire runtime based on the application of the new distribution scheme.

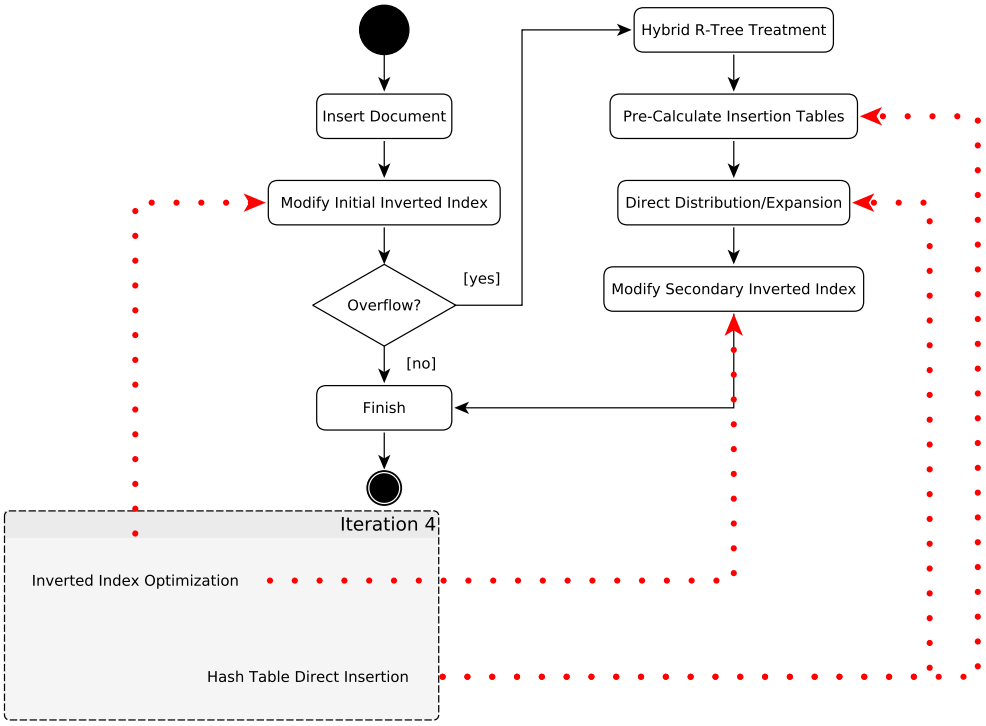
5.2.5.5 Summary

This iteration has reorganised the process of secondary inverted index manipulations. Besides these modifications, also the entire algorithm of distributing the items throughout the hybrid R-Tree has been altered. The latter does not at all affect the retrieval. Nevertheless, the treatment of elements which only occur in one document must be changed. As only the reference to this particular document is persisted directly inside the R-Tree leaf element afterwards, there must be an indicator for it, which must also be treated by the search process. However, this is the only part to be altered for getting the retrieval right again. The remaining parts of the software stay untouched. Most modifications applied here are only internal restructurings of the algorithms used for distribution.
The introduction of the bisection based searches in the B-Tree used to access the inverted index contents is directly notable. As expected, the quantity of comparisons lowers significantly. The combination of bulk operations on the secondary inverted index structures as well as the special

treatment of points occurring in only one document in conjunction with the inspection of sibling nodes is very beneficial for the runtime of the modifications. One row comparison does not take a lot of time. But, as the comparisons are issued very frequently, the total time required for them may be lowered a lot by reducing their total number. This is achieved by the combination of the bulk operations, the special treatment of points occurring only once and also the bisection based filtering, actions which reduce the total runtime tremendously. In addition, the distribution algorithm also changes entirely. While this does not have any influences on the retrieval algorithm because the structure of the access methods does not change, the total runtime lowers considerably. This results from the fact that the pre-calculated elements may be placed more directly and thus, not so many comparisons must be carried out. Comparisons always refer to the (de-)serialisation of tuples. They are already reduced by the utilisation of the respective caches. Yet, if the cache is full, "old" tuples are removed. Therefore, full (de-)serialisation operations must be performed which is bad for the runtime. Based on the more direct way of navigating to a leaf node where to place the pre-calculated tuples, these effects can finally be mitigated.
As only internal changes are performed, two iterations executed on the software are combined to one in this thesis.
Figure 5.79 shows the algorithm states before and after the iteration described in this section. Additional inverted index optimizations mainly refer to the initial as well as the secondary inverted index manipulation. The main changes are introduced in the management of seldom occurring elements. The first adoption is a special treatment applied to terms occurring exactly once.
The second main change is applied to the treatment of the hybrid R-Tree distribution. In this case, the entire algorithm for the manipulation of the augmented R-Tree is altered. In advance, first the R-Tree expansion phase is carried out which inserts all points from all affected document tuples into the R-Tree. As a first step, searches for the elements are executed and, if they cannot be found in the R-Tree, all points are placed in the hybrid R-Tree. The distribution of the document and term entries is deployed as a second step. This entire procedure is modified in the new version of the algorithm by pre-calculating the entries for the secondary inverted index structures. Hence, the hash table pre-calculation is utilised in conjunction with a more direct placement of the tuples to the hybrid R-Tree as well as the secondary inverted index in this iteration. The main changes are outlined in subfigure 5.79b whereas subfigure 5.79a shows the initial state of the algorithm before the adoptions.



(a) Before



(b) After

Figure 5.79: Activity Diagram of Changes Performed in Iteration 4

5.2.6 Spatial Structures for Spatial Distributions

The outcome of the previous iteration is that a pre-calculated set of values contributes to an efficient distribution of the items throughout the hybrid index. However, working with a map seems to be unsuccessful in this application domain. A map is an associative array which sorts the entries according to a hash value derived from the respective item to be stored. This hash is a representative value for the given item. It can be calculated from the item and should be identical for elements storing equal values. Hence, the items need not necessarily be the same ones but must equal each other. The hashing procedure has some drawbacks. On the one hand, a hashing function must be chosen which only produces few collisions. They lead to lists of items which are not identical to each other but share the same hash value. On the other, the items stored inside a hash map are sorted according to the hash function. Thus, no "natural" ordering is applied here. It is simply ordered by the results of some artificial function which is solely present to generate a representative value for a given item.
Yet, the entries to be distributed to the hybrid R-Tree are assigned spatially in (at least) one or two dimensions. Thus, additional knowledge about these elements already exists present in advance. Applying a simple hash function to these spatial elements destroys the potential spatial arrangement of the items to be distributed. Hence, the next step for this iteration which, again, summarises the results of two individual iterations is the introduction of a properly chosen spatial storage mechanism for temporarily allocated main memory management of point like values.
A KD-Tree [8] is selected for this task. Several strategies of the management and distribution of this structure are tested and evaluated to figure out a feasible method of navigating through the trees. In this case, the search inside the R-Tree for distribution purposes as well as the KD-Tree for the current temporary storage of items must be parallelised. Therefore, multiple combination possibilities need to be evaluated to find an improved version which fits to the requirements now.

Parameter	Value	Explanation
HLimit	200	Artificial Upper Bound
Element Count	5	Amount of Elements per R-Tree node
Document Count	2550	Amount of Documents to be inserted
Dataset	Wikipedia	Dataset to be processed (Wikipedia sorted by ids)
Split	R*-Tree Split	R-Tree splitting method
Choose Subtree	R*-Tree Choose Subtree	Method for Selecting a proper subtree to place an element in

Table 5.26: Settings for the Post-tests Test Suite Run

The chart of parameters used for the execution of the test suite for the runs is shown in table 5.26. Three extra runs are performed of the test suite based on the combination of two iterations for this step. Therefore, the number of repetitions of the test suite is fixed to 100, hence, the document count is set to $n = \lceil \frac{100}{2} \rceil \cdot 50 + \lfloor \frac{100}{2} \rfloor = 2550$ during the measurements. Three runs of the test suite are carried out for gathering the data. The first one represents the pre-tests of iteration 6 (or post-tests of iteration 5). Then a post-test run of iteration 6 follows serving as pre-test input of iteration 7 and finally the validation of iteration 7 is realised. To compare the results, the identical setup using the values given in table 5.26 is applied. Hence, the outcomes of all runs may directly be compared to each other.

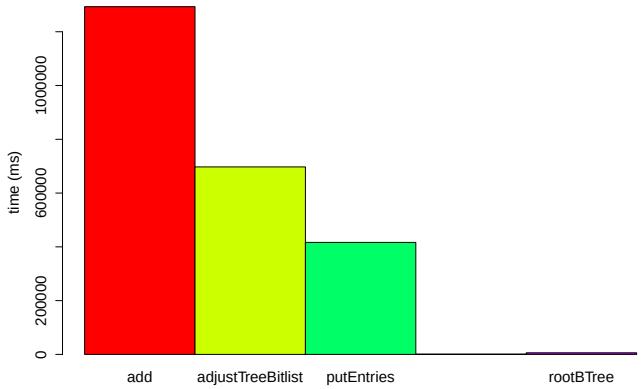


Figure 5.80: Duration of Phases during Pre-Test Run

5.2.6.1 Basic Observations

The duration of the respective phases can be depicted from figure 5.80. The entire process of adding items to the hybrid index is displayed in the `add` phase. A major difference to the preceding phases can be seen in the use of `adjustTreeBitlist` which is responsible for performing R-Tree related operations and adjusting the bitlists after the insertion of the respective items. Hence, this phase can be seen as an equivalent of the distribution in the previous iterations. It is most influential with respect to the entire runtime required. The phase of writing the entries to the final destination in the secondary inverted index structures (`putEntries`) comes next. The other two phases do not take comparably long. Thus, they may currently be left out of consideration.

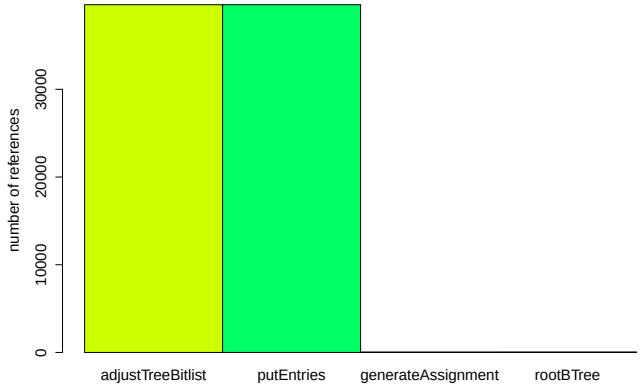


Figure 5.81: Number of References to the Respective Phases

The amount of references to the respective phases investigated can be seen in figure 5.81. This figure clarifies that `rootBtree` as well as `generateAssignment` are basically uninteresting for this phase because, on the one hand, they are not called very often and, on the other, do not require much time. `adjustTreeBitlist` is called as often as `putEntries` which is also obvious due to the current state of the implementation. For each entry in the assignment map the R-Tree is descended once, the entries are placed by means of `putEntries` and the references (at least the bitlists) are corrected using `adjustTreeBitlist` afterwards. Thus, these two methods are called in the same amount as they are always used as a pair of

executions. So, the average duration of `adjustTreeBitlist` is much higher than `putEntries`. Consequently, this phase needs to be optimised now. The total duration of `putEntries` is only ≈ 59.74% of the one of `adjustTreeBitlist`.

Function	Duration	Amount
adjustTreeBitlist	697, 260.667	39, 644
getParent	335, 567.154	742, 515
adjustSecondaryStructure	130, 589.575	247, 505

Table 5.27: Calls During `adjustTreeBitlist`

Table 5.27 shows the relationships of the phase `adjustTreeBitlist`. The method itself is displayed as a reference. It can easily be seen that retrieving the parent of one particular node consumes most of the time. As this method is recursively executed from a leaf node to the root, the process of getting access to a parental node is also required very frequently. However, the actual sense of this function is to adjust the bitlist. This is done in `adjustSecondaryStructure`, which only takes a small amount of the time compared to `getParent`. Hence, the administrative tasks take a lot more time than the actual task. As the actual work is supposed to be more time consuming than any administrative procedure, this is not the desired result.
The procedure executed in `putEntries` is also worthwhile investigating because it is the method used similarly often as `adjustTreeBitlist`. Still, it does not require the same amount of time to be executed.

Function	Duration	Amount
putEntries	416, 524.186	39, 644
insertInverted	250, 680.693	126, 871
findLeaf	159, 888.972	126, 871

Table 5.28: Calls During `putEntries`

The phase summary of `putEntries` as well as dependent function calls can be depicted from table 5.28. The phase of inserting the elements into the secondary inverted index is the one requiring most of the time here. This is obvious because the main task of adding elements to the B-Tree is fulfilled. The second most called and most time consuming method is the process of finding a proper leaf node. Besides requiring most of the time, these methods are also activated very often. This happens for each individual term id to be inserted at each particularly affected R-Tree element. The average duration per operation does not seem too bad. However, the frequent use of these methods, they require a lot of time in total.

Parameter	Value
Local Threshold	0.1
Global Threshold	0.1
Maximum Path Steps	3

Table 5.29: Parameter Values for Iteration 5

An individual analysis utilising the paths analyser tool is also executed with the parameter settings given in chart 5.29. This analysis results in exactly the same recommendations as already worked out previously. The main phases to be target of an optimisation are thus `putEntries` and `adjustTreeBitlist`.

5.2.6.2 Solution Alternatives

Two separate parts are selected for a potential optimisation in this iteration. The most important part, however, is the optimisation of the distribution of items throughout the tree because

according to the measurements it has the longest (average and total) duration. The placement strategy of elements in the inverted index has already been optimised in the preceding iteration(s) and, on the other, it does not have so much influence regarding the total runtime as the distribution mechanism.

5.2.6.2.1 Insertion of Elements to Secondary Inverted Index Structures The insertion of term id to document assignments is still a badly working part (regarding runtime) during the distribution of elements inside the hybrid R-Tree. This is not a direct result of the previous iterations. The currently used implementation is simply not yet efficient enough for proper use, although its performance has been increased enormously since the original implementation. On principal, only small changes are applicable in this case. As a direct observation of the basic analysis presented in section 5.2.6.1 the operation of inserting the entries is utilised $\approx 40,000$ times. This function then performs more than three times more insertions to the inverted index. As no efficient method of modifying the inverted index by applying internal optimizations is exists at the moment, another solution must be found to lower the quantity of uses for this functionality.
Multiple options are available at this stage:

1. Complete rebuild of B-Trees
2. Multiple separate trees
3. Piece-wise clustered insertions

The first option is a complete rebuild of the directory trees serving as access methods to secondary inverted index structures. It can be inspected from two sides. On the one hand, this option could improve the runtime if nearly the entire B-Tree is modified during one reorganisation. This seems to be promising because the entire B-Tree must be visited for insertion if the elements in the lists to be placed are distributed in each particular node of the tree. On the other, this option suffers from the situation that only a very small part of the B-Tree is affected. Therefore, it cannot be applied efficiently if only a very small portion of the B-Tree is concerned. Rebuilding the entire tree would result in a high work load without contributing a lot to the final result then.
Option 2 describes the situation where, e.g., a static part of an inverted index is set up. New items which are combined to a new B-Tree when, for example, the list length spills over a certain threshold are simply added to an overflowing list. Potentially, an additional B-Tree is set up then or the list is directly merged into the present B-Tree. This option seems to be encouraging. However, at the stage of secondary inverted index structures, an additional storage structure would be required to be designed which needs to be integrated to the existing algorithms and storage structures. This involves lots of changes in the present algorithms and also a large amount of additional storage.
Option 3 is a version checking for the possibility of inserting multiple items into one node. The items must be pre-processed to appear in the order they are likely to occur inside a B-Tree node. Thus, a natural ordering is applied as a first step to the items to be integrated. It already exists due to the construction of the assignments introduced inside the preceding optimisation iteration. The insertion operation may then take portions of the pre-calculated items to put them to the respective places. Thus, a node (and probably its sibling) may be filled directly with pre-calculated values. This makes bulk operations possible, at least on a node basis. Therefore, this strategy allows a rebuild of one node (and probably the siblings) and the main B-Tree operations do not need to be modified substantially. The basic intention of this change is to initiate bulk operations where possible but to affect only one node and potentially its following sibling.
These are the basic options one of which will be selected for implementation to enhance the reorganisation speed.

5.2.6.2.2 Adjustment of Bitlists The adjustment of bitlists is clearly the worst operation regarding the runtime in the reorganisation process of the hybrid index structure at the moment. Therefore, solutions must be found to overcome this issue. Hence, the current procedure responsible for distributing the items inside the hybrid R-Tree needs to be investigated.
The previous iteration introduced a hash map approach to pre-calculate the values to be inserted. It increased the performance tremendously. Yet, it exposed a new issue. The basic observations for this section are that the method `adjustTreeBitlist` is the most time consuming phase now. This function iterates recursively from a leaf node to the root node and sets bits inside the respective bitlist to indicate the presence of the terms given in the pre-calculated assignment list. Thus, it allows the process of re-finding elements inside the R-Tree based on the term id stored inside the bitlist. This is an important process to retrieve the data portions after storage. As the basic observations section shows, this procedure is mainly inefficient because of the frequent and long lasting calls to the function `getParent`, a method used to get access to the parent of the currently inspected node. It is utilised to recur iteratively upwards the tree in order to reorganise all elements on a certain path to a leaf node. Basically, this method performs everything properly. Yet, the basic question is how to reduce time, which mainly starts with the procedure of loading a respective parental node.
It must be noted that everything is stored inside a hash map in the current implementation state. This structure derives a unique identifier for an object based on a hash algorithm combining a structured value to a single one in a `long` representation. Depending on the assignment procedure, these `long` representations may also collide which means that two totally different objects with different contents share the same representative `long` value. It is also worth mentioning that the mapping function thus introduces an intrinsic order of the elements. The values memorised inside a hash map are stored inside an array. The position of each item there is calculated owing to the representative value originating from the object contents. Thus, an intrinsic order is given due to the derived values within an array. The process of generating and maintaining dynamic hash tables is quite old and has been investigated in detail (see, e.g., [66]). However, the addresses of the objects inside the hash tables seem to be assigned randomly based on the identifier and the length of the array to store them in. Yet, it is necessary to treat spatial objects during the distribution process of the lists to be inserted at the leaf elements of an R-Tree. The spatial assignment is totally ignored while the items items are placed utilising a hash table based approach. The basic cause for the high reorganisation cost of the `adjustTreeBitlist` phase can mainly be derived from the fact that one single path inside the R-Tree is hypothetically visited more than once because the spatial distribution of the items to be placed is not reflected in the storage of the dynamic hash table. Hence, a solution must be found respecting the spatial components of the items to be distributed throughout the R-Tree.
(At least) Two options are available to solve this issue:

1. Employ a space-filling curve
2. Implement a spatial structure for distributing the items

Option 1 refers to the introduction of a space-filling curve. There are multiple options for such a curve like the Hilbert or the z-curve. Generally, these curves use some kind of bit interleaving to map a value of multiple dimensions to exactly one. They are self repeating to fill the underlying data space entirely and arrange the memorised items in a spatially optimised way to retrieve them fast later on. The results of the multidimensional to one-dimensional mapping can then be stored inside an index structure for one-dimensional access. This approach is, e.g., utilised by the UB-Tree [83] which maps multiple dimensions into one and then saves the resulting datasets in a B-Tree.
Option 2 describes the possibility of implementing an explicit storage structure optimised for spatial access. Multiple variants could be used here. Obviously, one of these storage structures is the R-Tree used within the hybrid index. Yet, the R-Tree is originally intended for the use inside disk oriented storage environments. One of the storage structures optimised for range search access in main memory is the KD-Tree [8]. It is a binary tree separating the underlying data spaces in segments. This is achieved by recursively iterating to the particularly next splitting

dimension and separating the items in a "higher" and a "lower" part. Thus, each node has two direct children one of them containing elements lower (or equal) to the currently inspected coordinate value in the respective dimension and one comprising higher elements. Therefore, the KD-Tree supports multidimensional range searches for an unspecified number of dimensions separating the elements in two subtrees. Possibilities to pre-generate a KD-Tree on a static set of elements enhancing the retrieval to a logarithmic time complexity exist as well.
Thus, the decision regarding the adoption of the adjustment of bitlists must be made between these two variants of either setting up a space-filling curve or implementing a spatial access structure optimised for main memory usage.

5.2.6.3 Selected Solutions

This subsection outlines the selected solutions for the given issues of inserting elements to the secondary inverted index structures and the adjustment of bitlists. Intermediate steps are also inspected here because this section describes two individual iterations within the process of the thesis.

5.2.6.3.1 Insertion of Elements to Secondary Inverted Index Structures With respect to the insertion of elements to the secondary inverted index structures, a complete rebuild of the directory tree does not seem to be a good option. This version would be beneficial if the amount of elements inside the secondary inverted index was quite large and the amount of elements to be placed there was, too. Unfortunately, this cannot be guaranteed in general. However, there might be situations when this occurs. Depending on the numbers outlined in figure 5.72 in the preceding iteration, the quantity of references to one given point instance within the two inspected test corpora follows a Zipfian distribution. Hence, a low number of documents is pointed to by a high number of points inside the database table whereas a high amount of documents is referred to by a very low amount of points. Based on the Zipfian distribution of terms inside the documents as well, the same holds for terms to points in general. This indicates that the required high quantity of elements inside the term list assignments cannot be guaranteed in every case. The second option of holding multiple separate trees and re-combining them repeatedly or storing lists of elements containing unprocessed entries to be merged into a directory from time to time is also hard to setup because of the increased demand for additional management structures. Each actual directory B-Tree utilised to store an inverted index must be augmented by additional information to refer properly to these unprocessed lists or supplementary tree forks. Therefore, this option is also skipped here, at the moment. The third option instead looks the most promising, both regarding additional storage overhead and general applicability.
This solution is set up by pre-calculating items to be inserted to the secondary inverted index without having to load exactly the same path from the root node to a leaf repeatedly. As a first step, all items from the assignment list are converted to `SearchRow` entries, which may be further processed by the respective B-Trees. Yet, they are still kept in structures to retain equal items closely related to each other. This is done in order to manipulate these items using exactly the identical identifier (in this case a term id) at the same time. Hence, including them in a postings list or storing them directly inside a B-Tree leaf node can be decided directly without having to modify each of these items one after another. Yet, multiple different items may also be placed simultaneously if certain pre-conditions are fulfilled. This strategy can be used if the respective items fit in the same node as the first items to be inserted there or if the sibling node following the currently inspected one is supposed to store these items subsequently. Compared to the original B-Tree approaches, this procedure also keeps the implementation overhead low as well. Besides, it also reduces the possibility of having to load exactly the same path inside a secondary inverted index multiple times if more than one item may be placed inside one node (or the following sibling, potentially). Additionally, no adoptions need to be applied to the storage mechanisms as this change mainly refers to the algorithm and not to the underlying data structure.

5.2.6.3.2 Adjustment of Bitlists The adjustment of bitlists also comprises two solution alternatives. The first one is to optimise the generation of hash values using an adopted approach of generating the representative value for one key. Two important arguments can be brought in against this option. The first one is that a pre-defined implementation of the hash table is used in this work, which cannot be changed easily. Therefore, it is impossible to alter the internal mechanisms and a manually defined hash table needs to be created here. This does not seem to be very promising because the hash tables are generally introduced to find values using one pre-defined value in a short time. They are not suited very well for range searches. The internal distribution algorithm must distribute the entries from the initial assignment to R-Tree elements which, inside inner nodes, have spatial ranges defining a minimum and a maximum extension in each dimension. This also leads to the second issue of applying space-filling curves to hash tables. Normally, Hash tables are constructed by an array. It has a fixed capacity which may be enlarged over time. If an entry is inserted or searched for, its address is calculated by the representative value modulo the length of the array. This modulo operation tends to destroy the spatial order, potentially pre-generated by the derivation of the representative value for such a spatial structure. Based on these two issues, the implementation of a space-filling curve approach inside a hash table is skipped and the second option is selected for the optimisation of the adjustment of bitlists.
Due to the constraints given above, the decision is made to implement a data structure for the efficient management of spatial data in main memory instead of adopting the hash table based setup in order to apply spatial search strategies. Besides the hardly supportable storage of elements using a space-filling curve, a hash table is not well suited for spatial range queries. Thus, additional adoptions would have to be done to allow this type of queries. It is determined to utilise the KD-Tree [8] in the implementation here. Comparable to a binary tree, this tree structure separates the data space in two parts at each level. It is not a balanced tree. Yet, balancing can be achieved by utilising a fixed input set which is then recursively separated according to the dimensions and distributed to the subtree using always the median of the sorted lists as a split element.

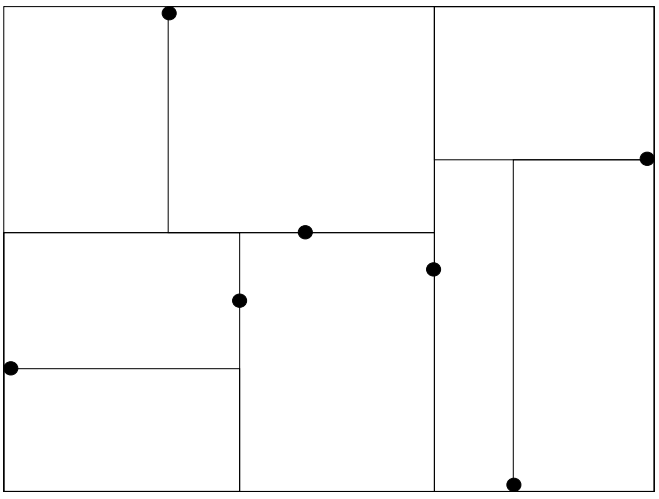


Figure 5.82: Spatial Distribution of Items inside a KD-Tree

A graphical conceptualisation of the spatial distributions inside a KD-Tree can be seen in figure 5.82. This tree structure is intended for the use in main memory and is thus suited for the given demand. Besides exact point searches, it also supports range searches in the respective elements efficiently. This KD-Tree is supposed to replace the hash table introduced in the preceding iteration. The distribution algorithm must only be changed slightly. Basically, the KD-Tree stores exactly the same assignments as the hash map introduced in the previous iteration. As the interface to the given structure is also quite similar, only a slight alteration of the distribution scheme is necessary. Besides these minor changes, all elements inside one node may now be treated at once as range searches within the KD-Tree can be handled efficiently,

too. Thus, all elements inside one leaf node are modified immediately, which limits the number of modifications on the identical node. This is basically the outcome of the first iterations examined. The only modifications regarding the distribution mechanism are:

- Introduction of a KD-Tree for efficient distribution of elements
- Use of range queries in leaf nodes to handle all elements inside one particular leaf simultaneously

Yet, it turned out that there might still be a potential for improving this procedure. Basically, only the performance issues at leaf nodes could be solved by using this approach. The inner nodes stay untouched by these changes. An amelioration becomes obvious if each leaf node is modified only once. Hence, one leaf node is not altered several times. Unfortunately, this does not apply to inner nodes. The same paths tend to be modified repeatedly.
To overcome this issue, the algorithm is essentially reverted to the initial state. The original distribution algorithm `distributeToSubtree` iterates recursively through the hybrid R-Tree, calculates the elements valid for a given entry and pushes the respective values to the subtrees. Thus, each node is visited only once if the pre-generated assignment contains elements valid at this subtree. Another possibility to overcome this issue would be a cache mechanism avoiding the need to load subtrees multiple times. However, the original distribution scheme seems to be more straight-forward.
Therefore, the original scheme is set up again here. Nevertheless, the performance gain is only small, here, which leads to another optimisation executed during this iteration. The original scheme examines each particular element and generates a list (or a KD-Tree after this iteration) of entries valid for this element. Two possibilities can be given for generating the entries valid for the subtree:

1. Construction of a subtree of the KD-Tree
2. Construction of a completely new tree

Unfortunately, neither of the potential solutions can be implemented efficiently because the construction of a completely new tree takes a lot of time if many entries are affected. The subtree generation is also a difficult task here because the KD-Tree does not support the introduction of subtrees, efficiently. The only possible solution is to pass the range valid for the subtree and, during retrieval, solely return elements which are inside this range. This approach does not seem to be constructive here. Hence, a third option is introduced.
In case of inner nodes, the KD-Tree simply serves as a comparison cache and at leaf nodes the actual entries are retrieved in order to insert them at the final destinations. As neither the construction of a subtree nor of an entirely new tree seems to be reasonable, the comparison cache simply checks whether at least one entry inside the KD-Tree exists that is valid for the currently inspected hybrid R-Tree element regarding the spatial component. The initially built KD-Tree is not modified at all until a leaf node is reached. At this stage, the respective elements valid for the given elements are extracted and inserted into the secondary inverted index structure. This procedure, however, makes the adoption of the bitlists for the inner elements impossible. Previously, this is implemented by inspecting the items which were inserted to the secondary inverted index structures and inherited to the root node recursively. As it is not known which entries of the assignment are valid at which subtree, this information must be gathered from another source. Either the bitlist valid for a given element could be generated by the combination of all children bitlists or the bitlists could be generated during the modifications and passed upwards the tree. The second possibility is applied now.
Therefore, the new algorithm starts (after the initial inverted index) with generating the assignments of point values to term ids pointing to documents. These assignments are pre-calculated in the form of a KD-Tree. Subsequently, the tree is recursively traversed. For each subtree, a subset of valid entries is not built any longer. It is simply checked if the subtree is a

potential candidate for further processing because at least one point of the initially built KD-Tree is inside the spatial range described by the element. If a leaf node is hit, the point elements inside the leaf node are retrieved from the KD-Tree and if the respective assignment lists are not empty, the secondary inverted index manipulation is executed. Afterwards, the term ids which were put aside are returned from the algorithm. This can be seen as a change list of term ids forming the bitlists to be set to the element. The union of the bitlists of each element contained inside a point is passed upward the tree and set there to reorganise the hybrid R-Tree properly again.

5.2.6.4 Evaluation

The evaluation of the results for this iteration is demonstrated in an overview of the three phases affected. Basically, the pre-tests are the results of iteration 5 which is followed by the intermediate step of iteration 6 and concluded with the seventh iteration. The table of settings used as an input for the test suite is already displayed in chart 5.26 and does not change at all because the parameters are not subject of discussion for this section.

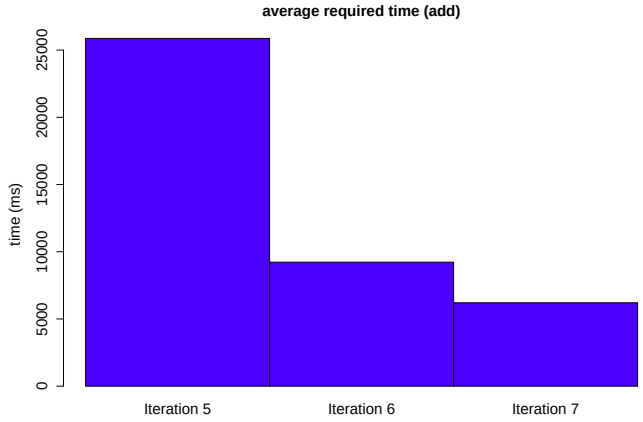
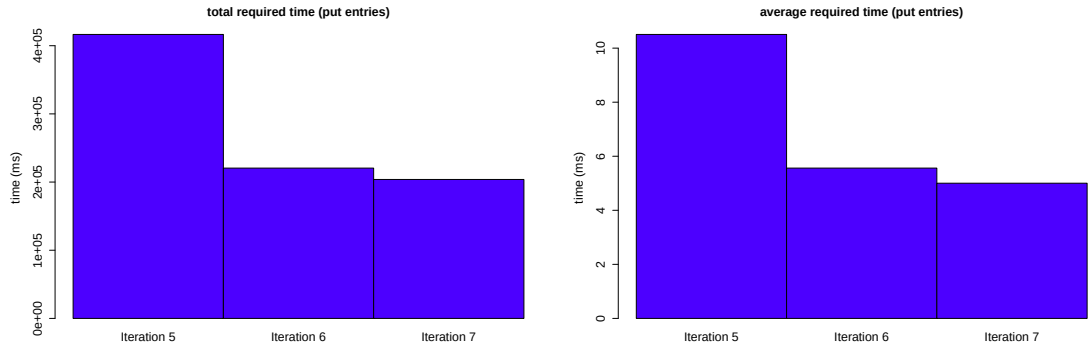


Figure 5.83: Add Function Comparison

An outline of the effects achieved with the given optimisation approaches can be depicted from 5.83. This figure shows the average runtime of the add function reflecting the entire reorganisation functionality. The time required by iteration 7 is only $\approx 23.99\%$ of the original one from iteration 5, which indicates that enormous effects regarding runtime are triggered due to the imposed changes of the iterations. However, it is probably worthwhile investigating the effects and the respective phases individually.

5.2.6.4.1 Secondary Inverted Index One of the main targets of this iteration is the optimisation of the runtime behaviour for the secondary inverted index manipulation algorithm. The times required for secondary inverted index manipulation can be depicted from figure 5.84. It shows that from iteration 5 to 6 the required time decreases remarkably whereas it only lowers slightly from iteration 6 to 7. This is probably obvious because the main changes to the algorithms are introduced in iteration 6. The small change from iteration 6 to 7 most probably results from side effects produced by the optimizations on the distribution functionality of iteration 7. The average times required follow the same rules as the total required ones, which indicates that the number of references to the respective phases does not change at all (or at least not much). This is also explainable because the functionality of the secondary inverted index manipulation is executed for each point entry in the assignment generated initially. Hence, as the same document set is inspected for all respective iterations under test, also the same

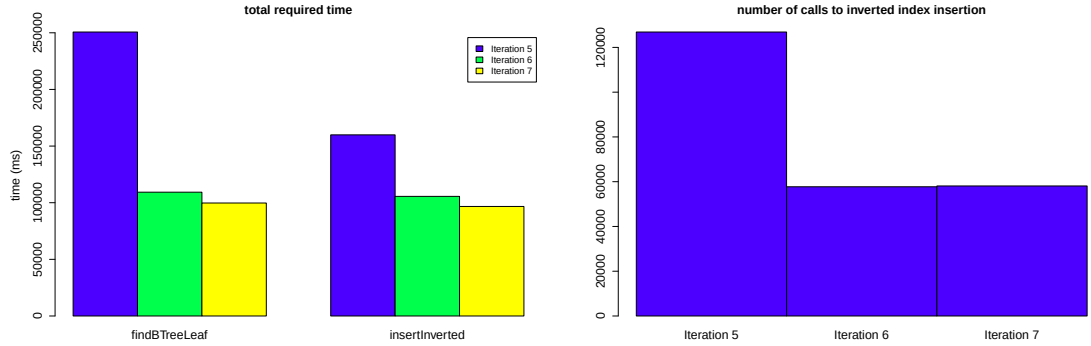


(a) Total Time Required

(b) Average Time Required

Figure 5.84: Secondary Inverted Index Modification Times

entries must be distributed, which leads to the situation that for each affected point the phase is executed once per reorganisation run.
The basic changes in the procedures are introduced due to internal modifications. The most important change affected in this case is the procedure of the actual inverted index manipulation which is the main time consuming method here.



(a) Required Time for Subphases

(b) Number of Calls to Phases

Figure 5.85: Statistical Values for Subphases of Secondary Inverted Index Manipulation

The statistical values for the given referenced subphases of `putEntries` in the respective iterations can be depicted from figure 5.85. The graphs show a tremendous improvement of the runtime of the respective phases, especially from iteration 5 to 6. The changes from iteration 6 to 7 are merely marginal. The basic cause for this improvement is the change in the treatment of multiple elements. Figure 5.85b demonstrates that the number of calls to the inverted index insertion phase, which is the most influential regarding runtime, can be lowered to $\approx 45.50\%$ decreasing the total runtime to $\approx 43.6\%$ of the original one from iteration 5. In total, the runtime is reduced to $\approx 39.78\%$ from iteration 5 to 7. As already noticed, the improvement between iteration 6 and 7 mainly result from side effects of the modifications of the distribution algorithm. The changes in runtime are mainly introduced by the differences of the number of references. Therefore, it can be stated that the application of bulk operations at this stage helps to reduce the total required time as, obviously, bulk operations performing multiple insertions inside one particular node and thus the ability to avoid the loading of exactly one path multiple times decrease the runtime significantly.
The average runtime required per insertion operation at one secondary inverted index is shown in figure 5.86. It can easily be depicted from this figure that the average runtime required slightly decreases for the insertion procedure on the secondary inverted index structure. However, the slight changes in the average runtime are not as big as the alterations in the total runtime.

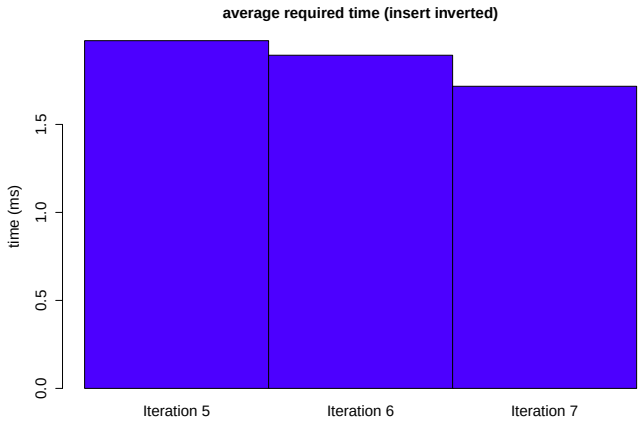
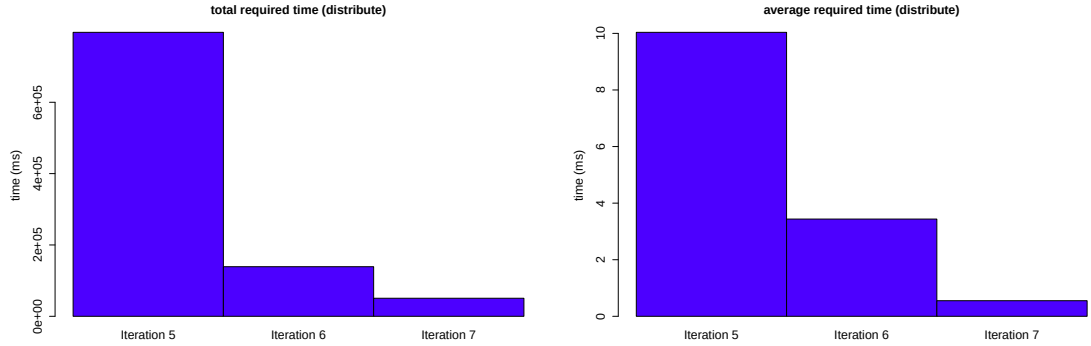


Figure 5.86: Average Runtime Required per Insertion

Hence, as the biggest modification in these figures is detectable in the quantity of calls to the respective methods and the average duration does not vary much, it is obvious that the major changes for the entire runtime result from lowering the number of executions of the secondary inverted index manipulation methods. This reduction can be derived from the introduction of bulk-like operations using the insertion scheme described in subsection 5.2.6.3.1.

5.2.6.4.2 Distribution The modifications implemented for the secondary inverted index insertions are only a smaller part based on the measurements of the basic observations. The more important part with the most influence on the runtime is the distribution of elements through the hybrid R-Tree. The effects of the inclusion of the KD-Tree as a spatial distribution method replacing the originally used hash table is discussed now.



(a) Total Time for Distribution

(b) Average Duration of Executions

Figure 5.87: Statistical Values for Distribution Subphases

The statistical values for evaluation purposes of the distribution adoptions is shown in figure 5.87. These graphics show that the total time required for the distribution lowers significantly. It must be noted that multiple methods are combined to one for the generation of these figures which have the main influence on the distribution of the elements through the hybrid R-Tree. Iteration 6 introduces the KD-Tree as a replacement for the hash table whereas iteration 7 includes improved methods for iterating through the tree without having to visit each node more than once. Visiting nodes in paths multiple times is potentially given in the initial version of the KD-Tree because it only blocks the accesses on the leaf level. The total duration of the distribution approach for iteration 6 is only $\approx 17.47\%$ of the one in the previous iteration. In

relation to iteration 6, the duration of iteration 7 is only $\approx 36.40\%$. Therefore, the total runtime after iteration 7 is $\approx 6.36\%$ of the initial one of iteration 5.
The differences of the average runtimes are even more precise in this case. Iteration 6 lowers the runtime to $\approx 34.31\%$ whereas the next iteration decreases to $\approx 10.17\%$ of iteration 6. This makes a total percentual runtime of $\approx 3.49\%$ of the seventh iteration compared to the fifth. Unfortunately, the distinct parts of the algorithms cannot be analysed or compared in detail because the entire algorithm has changed and thus, the comparison of individual parts can hardly be executed.

5.2.6.5 Summary

This section introduced two different parts for the algorithm, secondary inverted index bulk-like insertion operations as well as spatial main memory data structures for efficient distributions. Basically, solely internal algorithm parts have changed, which means that the search query parts need not be modified at all. Only the distribution and the manipulation of secondary inverted index structures are subject of change which does not involve any adoptions in the affected storage structures. Therefore, the search algorithms may stay the same as before.
The results which can be depicted from the secondary inverted index optimizations show that the bulk operations influences the runtime very much. Besides, this also shows that these optimised operations can be executed very often, which could have two potential reasons. On the one hand, the secondary inverted index structures could be very small and, on the other, the case of multiple items placed in one node (or the sibling) occurs frequently. Most probably, a combination of these two causes leads to the improved runtime of the inspected algorithm.
The main focus of this optimisation iteration has been the adoption of spatial structures for an improved distribution behaviour. The introduction of the KD-Tree and the alteration back to the original distribution scheme using a cache lookup behaviour perform very well now. One major change is obviously the implementation of the KD-Tree, which rises the performance tremendously compared to a hash table, because it is well suited for point and range searches. Hence, the generation of lists can be performed straight-forward based on the pre-calculated buckets to be placed to the secondary inverted index structures. However, in the sixth iteration the performance could still be increased. On principle, two options exist here. One is the introduction of a caching mechanism for already visited nodes, which skips the necessity to load one node multiple times during the traversal. This option has been cancelled in favour of the initial algorithm which recursively iterates through the tree and distributes items where they spatially fit. It benefits from only very local modifications of the R-Tree elements. It still implements a depth-first search strategy on the R-Tree elements but has the ability to skip the requirement of having to re-load an R-Tree page if no modifications have been performed on lower levels. The measurement data justify this result and state that pre-calculating spatial main memory data structures increase the performance for the distribution of spatial data.
The adoptions carried out in this iteration can be graphically depicted from figure 5.88. In comparison to the previous iteration (see figure 5.79), the algorithm has not changed much. The main difference in the hybrid R-Tree modification lies in the introduction of the KD-Tree which replaces hash table realised in the previous iteration. The evaluation shows that the spatial data contained in the KD-Tree for distribution purposes are much better suited to the requirements given for checking the necessity of descending into a certain subtree of the R-Tree. Additionally, bulk operations are introduced for the modification of the secondary inverted index structure. These bulk operations enable the insertion of multiple elements to the inverted index structure if they can be placed in the currently inspected target node or in one of its siblings.

This iteration closes the discussion about optimisation approaches for the efficient reorganisation of hybrid index structures supporting multimedia search criteria. The preceding sections describe an integrated and structured approach for the improvement process of the algorithms carried out on secondary memory. The remaining parts of the thesis delineate the finalisation and summary of the work carried out. Furthermore, additional measurements are performed to clarify issues not handled in the preceding chapters.

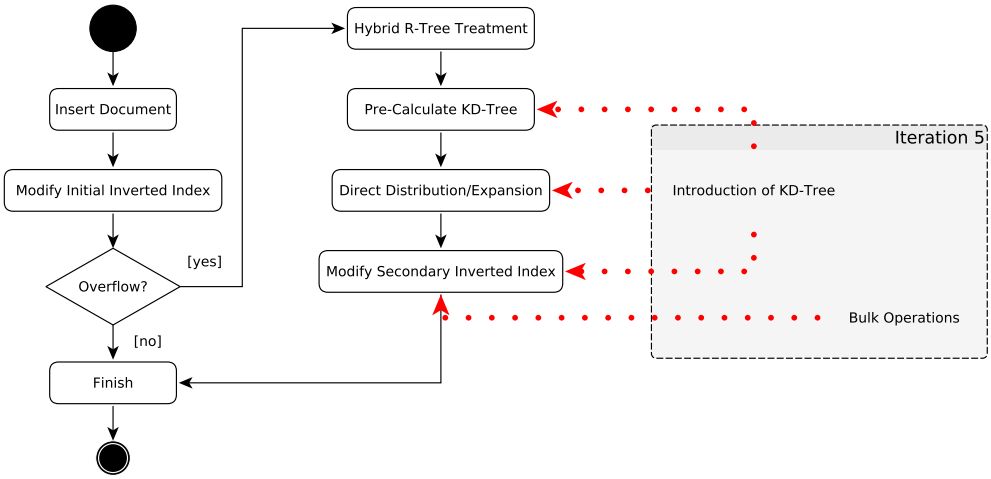


Figure 5.88: Activity Diagram of Changes Performed in Iteration 5

6 FINAL RESULTS AND EVALUATION OF THE OPTIMISATION

In the following, the final results of the optimizations performed in this work are described. Additionally, an evaluation of the entire results is carried out. Thus, also a survey of all optimisation iterations is given, which refers back to the respective sections. One of the issues of this thesis, i.e., the difference of performance for the various corpora used, is also clarified based on experimental validations. This leads to a restructuring of the document order and a final validation of the performance.
This chapter actually outlines the final state of the access structure as well additional measurements. The final deliverables are re-evaluated in a study providing an overview of all iterations carried out in chapter 5. This is described in section 6.4. In addition, the final algorithm state is listed in section 6.1. Both of these tasks refer to the description of the artifacts produced in this research work. In total, the main target of this chapter is the evaluation of the design tasks performed during the reorganisation optimizations in chapter 5. It refers to the design evaluation guideline (see 1.2.2.1.3). In addition, it lists, with exception of the model, also the basic research contributions of this thesis (see 1.2.2.1.4).

6.1 FINAL STATE OF THE HYBRID INDEX ALGORITHMS

The final state of the hybrid index algorithms is depicted hereafter listed in pseudo-code to outline the most important parts. Besides, also the major changes introduced in the respective optimisation iterations are outlined. Thus, references to all parts of the thesis up to now exist, especially to chapter 5, which introduced the optimizations.
This section is separated by the respective functionalities (insert, delete and find) of the hybrid access structure. Furthermore subsections are given for the particular storage structures affected by the changes.
The pseudo-code descriptions use a Java- / C-like syntax for the internal descriptions.
The central results including some of the algorithms contained in this section can be found in [63].

6.1.1 Delete

The deletion of elements is not much inspected in the current state of the hybrid access structure. Basically, it is assumed that the hybrid access structure is instantiated inside an information system or in a web search engine supporting enterprise content management or geographic information retrieval.
Documents managed in a geographic information retrieval oriented application do not tend to be removed or updated. Once they have been added to the storage structure in a web search engine and all properties to be sought for are derived from the textual documents, they are not changed any more. Therefore, a detailed deletion algorithm is skipped here.

Algorithm 6.1: Delete Document

```
1 Function Delete(document) /* algorithm for deletion                    */
    Data: the item to be deleted as found by the retrieval mechanism
    /* simply delete the item from the document heap and decrement
       the row count                                                      */
2   documentHeap.deleteDoc(document.getKey());
3   decrementRowCount();
```

The algorithm to remove a document from the hybrid index can be seen in algorithm 6.1. On principal, this algorithm deletes the document just by invalidating the row in the document heap which is used to manage the document references and decrements the row count. So, the index structure to conform to the requirements given by the H2 database is made possible.
A more sophisticated version of the deletion algorithm would iterate through all affected storage structures (initial inverted index, hybrid R-Tree, secondary inverted index and document heap) and remove all references to this particular element. However, this is omitted here. The retrieval algorithm would then find a valid document and try to return it to the user. After applying the deletion algorithm, the element is set to invalid inside the document heap. Hence, this item is skipped and the retrieval algorithm continues searching for the next one. Consequently, this approach is sufficient for the requirements here.

6.1.2 Add

The insertion algorithm is more complex than the deletion. Due to the fact that this part of the index structure is the main target of investigation for this thesis, it is described in more detail. At the beginning the general approach by means of the initial hybrid index structure is outlined. It refers to the respectively modified storage structures which will subsequently be described.

Algorithm 6.2: Insert Document (adopted from [63])

```
1 Function HybridIndex::Add(document) /* algorithm for insertion       */
    Data: the item to be inserted to the hybrid index
2   documentHeap.Place(document.getPoints());
3   toDistribute = initialInvertedIndex.Insert(document.getWords());
4   if toDistribute != ∅ then
5     hybridRTree.Insert(toDistribute);
```

Algorithm 6.2 delineates the insertion process on the hybrid index. First, the point references stored inside the document are distributed to the document heap. This storage structure is basically a linked list which appends the respective points at the end of the pages. If the required space of one point array instance is too high, the respective array is split in multiple parts to optimally utilise the available space.

The next part is the insertion of the terms to the initial inverted index. The initial inverted index handles the overflow treatment of terms occurring with a high frequency, which is determined by *HLimit*. Hence, the list of entries returned from `initialInvertedIndex.insert(...)` contains all elements either overflowing this limit for the first time or previously marked as high frequently used. These entries must then be managed by the `hybridRTree` insertion functionality, which performs the main tasks of distributing these entries through the R-Tree augmented with bitlists. The individual algorithms for the respective structures are discussed in the subsequent subsections.

6.1.2.1 Initial Inverted Index

The inverted index structures used in this thesis are accessed via a B-Tree. Hence, the sorting and storage management is executed via B-Tree operations.

Algorithm 6.3: Insertion at Initial Inverted Index (adopted from [63])

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

The algorithm applied to the non-normalised elements (called 'terms' in this terminology) to insert them to the initial inverted index can be seen in algorithm 6.3. Generally, for each term a lookup in the inverted index is performed. After that, a check is executed to find out whether the element is already present and if it potentially already overflows the limit. That means that, in this case, the term id which is used as bit index is subsequently set. If this is true, the element is directly added to the set of overflowing entries. Otherwise, the element is inserted by using the inverted index insertion procedure described in algorithm 6.4. This algorithm returns the reference count representing the term frequency. If it is higher than the specified *HLimit*, a new item id is generated by incrementing the one currently present. Afterwards, the item is replaced by removing all but one references and setting the term id to indicate that the item is shifted to the hybrid R-Tree. Additionally, the term including all document references from this term is also added to the overflowing list which is finally returned.
The algorithm for inverted index modification is displayed in algorithm 6.4. It is also used in the secondary inverted index structures and is introduced due to the results of iteration 5.2.4. First, a check for the current storage strategy is executed. If the item is stored internally in B-Tree nodes, another validation is performed to detect whether the currently stored items and the new element, additionally, fit into one page. Otherwise, the elements are moved to a so-called inverted page which manages all references of one term to the respective documents. If the additional element still fits into the page, it is directly placed there. In case the element is already stored in an inverted page, the new element is added there. During the respective

Algorithm 6.4: Insertion on Inverted Index (adopted from [63])

```
1  Function InvertedIndex::Insert(node, element) /* algorithm for inverted
     index modification                                                  */
       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 ! invertedIndex.CheckSize(node, element) then
4              invertedIndex.MoveToExternalPage(items);
5              invertedIndex.PlaceToExternalPage(element);
6          else
7              invertedIndex.PlaceToNode(element);
8      else
9          invertedIndex.PlaceToExternalPage(element);
10     return frequency
```

operations, the frequency of the element is collected and returned to the caller to determine whether the frequency is higher or lower than *HLimit* subsequently. The frequency of items is only interesting at this stage and is ignored in the remaining algorithms like the secondary inverted index manipulation.

6.1.2.2 Hybrid R-Tree

The algorithm most changed in the hybrid index processes is the manipulation of the hybrid R-Tree.

Algorithm 6.5: Hybrid R-Tree Insertion (adopted from [63])

```
1  Function HybridRTree::Insert(overflow) /* hybrid R-Tree insertion    */
       Data: the overflowing list to be distributed
2      kdtree = hybridRTree.GenerateAssignment(overflow);
3      hybridRTree.Distribute(kdtree, root);
4      hybridRTree.PlaceRemaining(kdtree);
```

Algorithm 6.5 gives an overview of the hybrid R-Tree insertion. The first step is to generate the assignment. This assignment consists of the KD-Tree, which is a direct result of the pre-calculation (see section 5.2.5), and the introduction of spatial distributions (see section 5.2.6). It must be noted that each of the respective functions utilising items from the hybrid R-Tree and the secondary inverted index (and the initial inverted index, partially), directly profit from the introduction of the cache mechanisms (see section 5.2.2) and the changes in the storage structures (see section 5.2.4).
The pre-calculated KD-Tree assignments are then passed to the distribution algorithm (see algorithm 6.6) which performs the task of distributing the elements to "fitting" places inside the tree and also modifies the secondary inverted index. The final step is to place the remaining entries. The initial distribution is only executed on elements (points in this case) which are already present inside the hybrid R-Tree. Thus, new points not contained in the R-Tree so far must be placed inside the tree, as well. Hence, each of the remaining points is inserted to the R-Tree by means of the `placeRemaining` procedure and the respective secondary inverted index structures are created there, too. The insertion at the R-Tree is performed by using the results of section 5.2.3. For this reason the R*-Tree splitting and choose subtree metrics are used in the default settings here.

Algorithm 6.6: Distribution of Elements (adopted from [63])

```
1  Function HybridRTree::Distribute(kdtree, node) /* distribution of
   elements                                                          */
       Data: the KD-Tree assignment and the node to be modified
2      for entry ∈ node do
3          if ! isLeaf then
4              if kDTree.CacheHit(entry) then
5                  subtree = hybridRTree.LoadSubtree(entry);
6                  hybridRTree.Distribute(kdtree, subtree);
7          else
8              assignment = kDTree.GenerateCandidates(entry, kdtree);
9              secondaryInvertedIndex.PutEntries(entry, assignment);
10     hybridRTree.UpdateBitlist();
```

The distribution algorithm which has changed multiple times can be seen in algorithm 6.6. The final version is introduced in section 5.2.6 due to the introduction of the KD-Tree as a spatial storage mechanism for main memory. A distinction is done regarding the leaf state of the currently inspected node. In case of an inner node, a check is performed to find out whether the range representing the spatial component for the given element contains at least one entry inside the KD-Tree. Thus, this element is a candidate for distribution. Provided this is true, the subtree is loaded and the distribution procedure is executed recursively. If a leaf node is reached (see `else` branch, line 7 ff.), the point is searched inside the KD-Tree and the assignment is generated by loading the list of entries assigned to this point from the KD-Tree. Subsequently, these elements are inserted to the secondary inverted index using the `putEntries` algorithm. Finally, all bitlists of all entries contained in the currently inspected node are updated to refer to the right portions stored below the given node.

6.1.2.3 Secondary Inverted Index

The secondary inverted index manipulation is also of special interest for this thesis. It has been optimised in multiple phases.

Algorithm 6.7: Insertion Operation on Secondary Inverted Index (adopted from [63])

```
1  Function SecondaryInvertedIndex::PutEntries(entry, assignment)
   /* insertion of elements inside the secondary inverted index     */
       Data: the hybrid R-Tree entry and the assignment to be placed
2      if secondaryInvertedIndex.CheckSingleDocument() then
3          secondaryInvertedIndex.SetRowKey(document);
4      else
5          if row key not empty then
6              secondaryInvertedIndex.MoveToSecondary(entries);
7          while assignment != ∅ do
8              toAdd = secondaryInvertedIndex.CalculateEntries(assignment);
9              invertedIndex.Insert(toAdd);
```

Algorithm 6.7 describes the operations on the secondary inverted index. The first step is to check if only one single document is contained inside the assignment and whether the secondary inverted index only comprises references to this document (or is not present, yet). In this case, the row key is set to the hybrid R-Tree element to point to this particular document.

This change is introduced in section 5.2.5 because many point values only occur in solely one document. Hence, the instantiation of an external secondary inverted index in the form of a B-Tree can be omitted in a lot of instances. In the parallel branch (`else` lines 4 ff.) references to more than one document exist. If already (at least) one document is referenced there, it is first moved to an external secondary inverted index; subsequently the algorithm may proceed. The next step starts with a loop until the entire assignment has been processed. First, all entries which may be placed at one node (or its next sibling) are calculated. They are subsequently put by using the inverted index manipulation algorithm applying a distinction between commonly used items which are placed to inverted pages or seldom used items placed directly inside the B-Tree leaf nodes. The bulk-loading inspired operations result from section 5.2.6. The distinction of strategies (LEAF and EXTRA combined to HYBRID) is introduced in section 5.2.4 and already described in algorithm 6.4.

These descriptions are the main algorithms used for the hybrid index structure. Besides there are obviously implementations for all affected storage structures (B-Tree, R-Tree, inverted index and additional management structures), too which are omitted here because they follow the standard metrics demonstrated in the original proposals.

6.1.3 Find

The basic retrieval concept of the H2 database relies on a cursor concept. A cursor is an approach to fetch results iteratively from the database. Hence, at each query to the database, first a cursor is constructed which is then worked on. The most fundamental operations to support are the check to detect if further results exist as well as the retrieval of the particular results. Therefore, the access structures must make successive retrievals possible. Thus, mechanisms are necessary which ensure that, on the one hand, not very much main memory is wasted and, on the other, the retrieval is facilitated efficiently. A proper storage mechanism is required for this task.
Basically, the main problem with this approach is to prepare the data properly for this kind of retrieval. In the case of the hybrid index, it is ensured that the document references are stored in a sorted way. This is done by assigning higher internal item ids (referring to the document heap) when storing the elements. Therefore, in each case, allocations to new pages are carried out at the end of the database file. By means of this, new document elements are always stored at the end and thus get higher identificators. So, a sorting mechanism of the items in all levels of the hybrid tree is feasible.
A well-known possibility for the retrieval of the top-k elements in a sorted set is the no random access algorithm [28]. The given algorithm does not at all require information about the entire dataset [52, pages: 11:14 – 11:17]. The intersection of the postings lists in the final step of the retrieval is a task potentially demanding a lot of main memory. By using the no random access metrics, this amount may be limited. Hence, the document ids inside the document heap are assigned sequentially growing. So, an adopted no random access approach can work for the retrieval.
The cursor concept is omitted for the description of the algorithms here.

Algorithm 6.8: Retrieval of Elements

```
1 Function HybridIndex::Find(row) /* retrieval for the given row to be
  found                                                              */
    Data: a template row containing the set of terms and a range to be found
    Result: the result set for the search criteria
2   (bitlist, results) = initialInvertedIndex.Find(row);
3   if bitlist != ∅ then
4     results = hybridRTree.FindLeaf(row, bitlist);
5   return results;
```

The retrieval algorithm (6.8) consists of two parts. First, the search is executed on the initial inverted index. Depending on the pair of results returned from there, the algorithm decides whether to continue inside the hybrid part of the index or to return the results directly. If the bitlist is not empty, the search continues otherwise and the results are directly returned. The circumstances leading to these conditions are outlined in algorithm 6.9.

Algorithm 6.9: Retrieval of Elements from Initial Inverted Index

```
1  Function InitialInvertedIndex::Find(row) /* search inside the initial
   inverted index                                                    */
       Data: a template row containing the search predicates to be found
       Result: a pair of a bitlist and results from the initial inverted index
2      bitlist = ∅;
3      results = ∅;
4      for term ∈ row.GetTerms() do
5          invertedIndex.Find(term);
6          if ! found then
7              return (∅, ∅);
8          else
9              if term id is set then
10                 bitlist = bitlist ∪ termId;
11             else
12                 if results == ∅ then
13                     results = results ∪ result;
14                 else
15                     results = results ∩ result;
16                 if result == ∅ then
17                     return (∅, ∅);
18     return (bitlist, results);
```

Algorithm 6.9 describes the search procedure of the initial inverted index. The algorithm takes each term of the given terms to be found in the template row and looks it up in the inverted index. If one of the terms is not found, a pair of empty sets is returned indicating that no results are found for the template row. However, if each term is found, multiple paths are possible. In case a term id is set for the found term, the bitlist is constructed as the union of all term ids. This path is taken if the term frequency is higher than *HLimit*. The other branch is chosen if the term frequency is lower than *HLimit*. Then, the results for this term are given directly inside the initial inverted index. Hence, the spatial references are retrieved from the document heap and checked for containment in the search range. These intermediate results are either joined to the outcomes using a union operation if the result set is initially empty or intersected with the already existing ones. If the final output of the union or intersection is empty, the algorithm notifies the calling function (see 6.8) saying that no findings for the given query are available and finishes.

The next stage of the search algorithms is the retrieval inside the hybrid R-Tree. It is closely oriented at the retrieval of the R-Tree and checks the bitlist additionally.

Algorithm 6.10 describes the search for a valid leaf element inside the hybrid R-Tree. It iterates through the elements of one particular node. First, the bitlist is checked to find out whether the respective bits representing the search terms determined in the initial inverted index are set for the respective element. Subsequently, a distinction is executed for leaf and inner nodes. In the case of inner nodes, the overlap of the stored element range is compared with the search range. If an overlap occurs, the subtree is loaded and the search continues there. For leaf nodes, a check is executed to see whether the search range includes the point stored inside the element. Provided this is true, the element is returned as a candidate. This search procedure ensures that only elements are kept if the bits are set as well as the spatial criterion is fulfilled.

Algorithm 6.10: Find a Leaf Node inside the Hybrid R-Tree

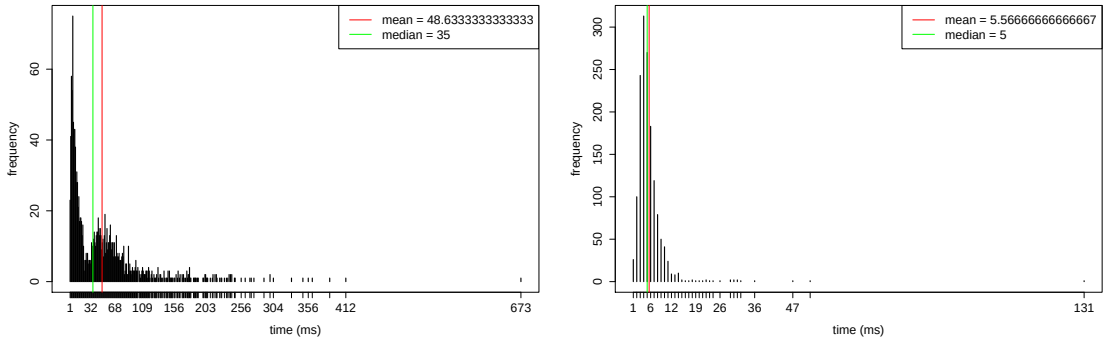
```
1  Function HybridRTree::FindLeaf(bitlist, range, node) /* search inside the
   hybrid R-Tree                                                          */
       Data: the bitlist generated by the initial inverted index as well as the search range to be
             used as spatial criterion and the currently inspected node
       Result: a leaf element fulfilling the search criteria
2      for element ∈ node do
3          if ! bitlist.Contains(element) then
4              continue;
5          if leaf then
6              if range.Contains(element) then
7                  return element;
8          else
9              if range.Overlaps(element) then
10                 subtree = hybridRTree.LoadNode(element);
11                 leaf = hybridRTree.FindLeaf(bitlist, range, subtree);
12                 return leaf;
```

The last step in the retrieval is the search in the secondary inverted index. This algorithm applies the no random access strategy. Two cases are initially given for the secondary inverted index structures. The first one refers to the case when only one document is valid for all term ids at one point. This is handled by an entry directly in the leaf element. In all other cases, inverted index searches are executed where each of the terms is either stored by using the LEAF or by applying the EXTRA strategy. The basic search procedure follows the inverted index search by parallel lookups of all terms and comparisons in a no random access like strategy.

It must be noted that all search strategies must be implemented by using many internal states. This is necessary to ensure that the elements can be retrieved one after another. Hence, the cursor may ask for the particular next item. Therefore, the state must be kept updated to make this behaviour possible.
Note, that parts of the algorithm descriptions as well as the main results from this sections may also be found in [63].

6.2 COMPARISON OF CORPORA REGARDING REORGANISATION PERFORMANCE

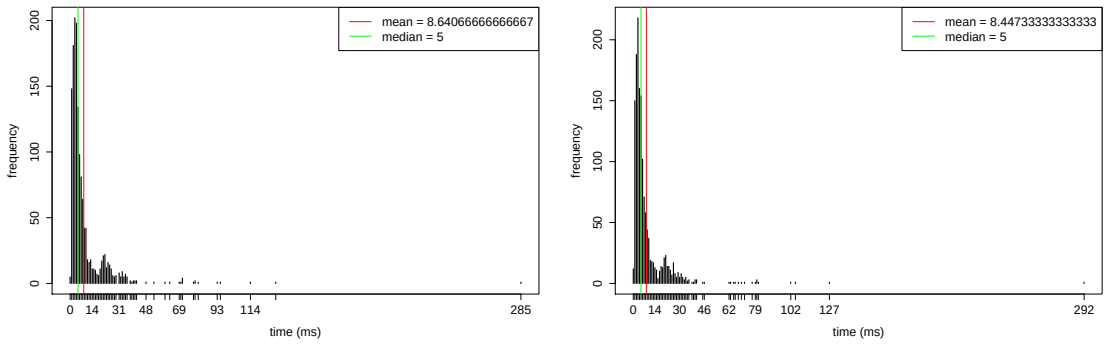
The entire optimisation iterations are executed on the Wikipedia corpus. For comparison and evaluation purposes, also the Reuters corpus was taken into account. Already in a very early stage of the project it was noticed that the performance of the reorganisation algorithms using the Reuters corpus is far better than the one of the Wikipedia corpus. This is, probably not a very obvious result. At least the mean and median given in the analysis chapter (see sections 5.1.3.2 and 5.1.3.3) do not differ significantly. All of them are quite similar to each other. At least they do not differ as much as the performance for the reorganisations. Even after the last optimisation iteration, there are still tremendous differences between the two corpora.
To give a short impression of the times, a test case is executed on $1,500$ documents for both datasets. The results with respect to the reorganisation times of these dataset can be depicted from figure 6.1. Besides the actual times, it also shows the arithmetic mean and the median time required for the set of instances. It can be seen that the values for insertion of Wikipedia documents tend to be much higher (factor 7 to 9). The basic question is why there are such tremendous differences even for this small quantity of documents inspected here.



(a) Wikipedia Dataset (b) Reuters Dataset

Figure 6.1: Times and Frequencies for Wikipedia and Reuters

Generally, as already demonstrated in sections 5.1.3.2 and 5.1.3.3, at least the average values (mean or median) for text lengths and the number of points per document do not differ much. Additionally, even for Heaps' and Zipf's law, there are no big differences between the two corpora.
Thus, as the general properties of the corpora do not differ much, it is worthwhile investigating the respectively used document instances. As a preparatory step, each of the corpora was split into multiple parts. This was done to be able to take parts of the documents as training and others as test data whereas this metric was not used in this thesis. For each corpus five different parts were split to be able to apply different amounts as training or test data. Each of the corpus parts comprises the same quantity of documents. Besides the comparisons of test data for the different data sets (Wikipedia and Reuters), also tests on the different parts of the corpora were executed.



(a) Wikipedia 2nd Portion (b) Wikipedia 3rd Portion

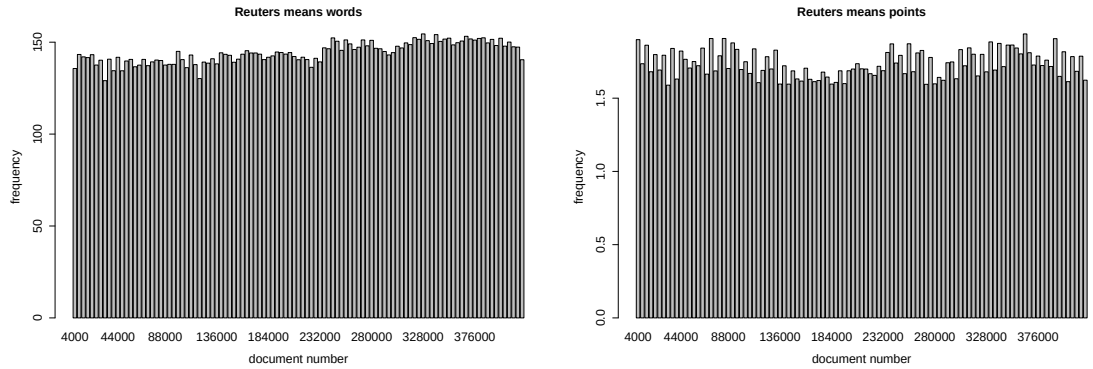
Figure 6.2: Times and Frequencies for Wikipedia Portion two and three

The two subfigures of figure 6.2 show the numbers for the second and third portion of the entire Wikipedia dataset. The first figure (6.1a) results from the first portion at the beginning of the file. Obviously, the median of the values is the same as the one of the Reuters data. The average value is slightly higher in these cases. The question which arises here why these temporal differences. The outer circumstances such as the parameters for the tests have not changed at all. Additionally, the tests are executed multiple times to avoid influences of temporal processes inside the executing PC. Even in repeated runs, the same figures could be obtained. Hence, it is probably worthwhile investigating the differences between the files more precisely. Therefore, the file sizes are looked at first.
Table 6.1 shows the file portions and the sizes of the files on the hard disk. It is obvious that the first part is significantly bigger regarding the necessary storage space than the following ones.

File Index	Size
1	1.3*GB*
2	836*MB*
3	689*MB*
4	647*MB*
5	650*MB*

Table 6.1: File Sizes of Wikipedia Portions

The second one requires only $\approx 63.9\%$ of the space of the first one whereas the third one (and the following ones) needs only $\approx 52.69\%$ (or less) of the first one. The question is where these differences come from as each of the respective files contains the same quantity of document instances but the file size on the disk differs significantly. These numbers obviously show that the differences between the corpora are based on the file contents. Therefore, a closer look is taken at the actual contents of the files and corpora.
An analysis regarding the number of terms per document as well as the number of points per document is carried out. Average values for these parameters are taken to summarise specific portions of the files and to give an overview of the documents. The original sequence of the files stays as given from the initial parsing and the analysis of the documents. Hence, the summarization is performed by building mean values of the number of terms and points per document. $4,000$ document instances are inspected at a time and the mean values are built of them.

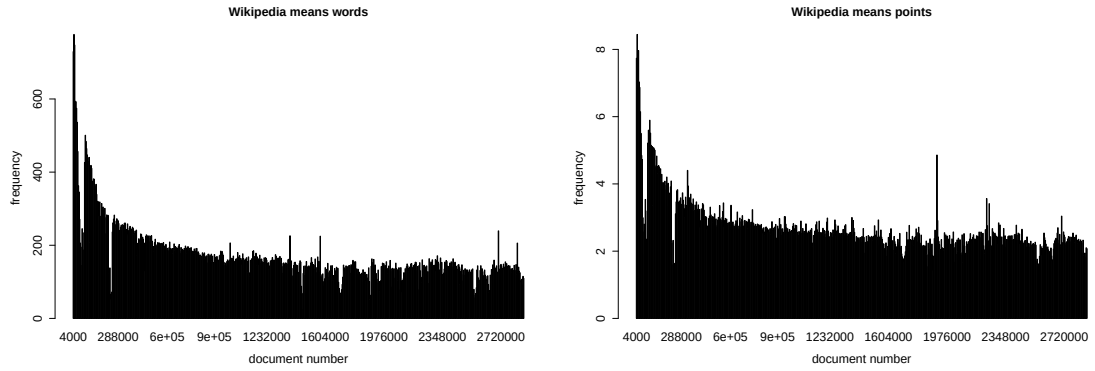


(a) Mean Words

(b) Mean Points

Figure 6.3: Mean Values for Number of Terms and Number of Points (Reuters Dataset)

The mean numbers for the Reuters dataset are shown in figure 6.3. It demonstrates that the average numbers for mean words as well as mean points do not vary much between the respective portions each of which represents a collection of $4,000$ documents. Yet, the Wikipedia dataset displays a totally different behaviour.
The same parameters as for the Reuters dataset are applied to the Wikipedia corpus. The results of this analysis are presented in figure 6.4. This analysis shows the outcomes of taking mean values of the analysed Wikipedia articles and applying them to $4,000$ documents each. It can be seen that for the first instances the number of terms per document is very high (≈ 800). This directly correlates with the average number of points for each document. These numbers decrease rapidly with the amount of inspected documents. Thus, for the documents starting with index $\approx 256,000$, the value is already only $\approx \frac{1}{3}$ of the initial one. For executing the test runs during the actual work, the first portion which contains the biggest documents regarding the amount of terms and points, has always been taken.
The question arising now is why exactly this distribution of text lengths exists inside the Wikipedia articles. It must be noted that it is impossible to give an unambiguous answer here. Probably, an attempt for the explanation can be made. Hadoop, which is used for the parallel extraction and analysis of the articles in both cases for Reuters as well as Wikipedia, performs

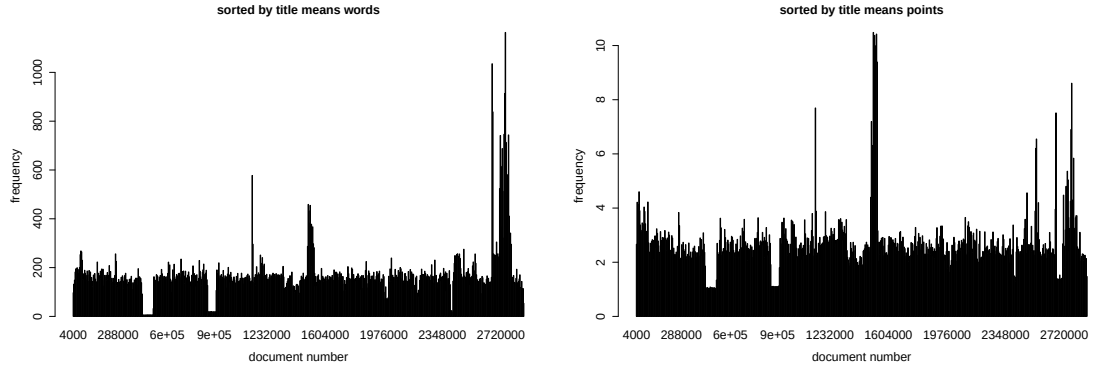


(a) Mean Words

(b) Mean Points

Figure 6.4: Mean Values for Number of Terms and Number of Points (Wikipedia Dataset)

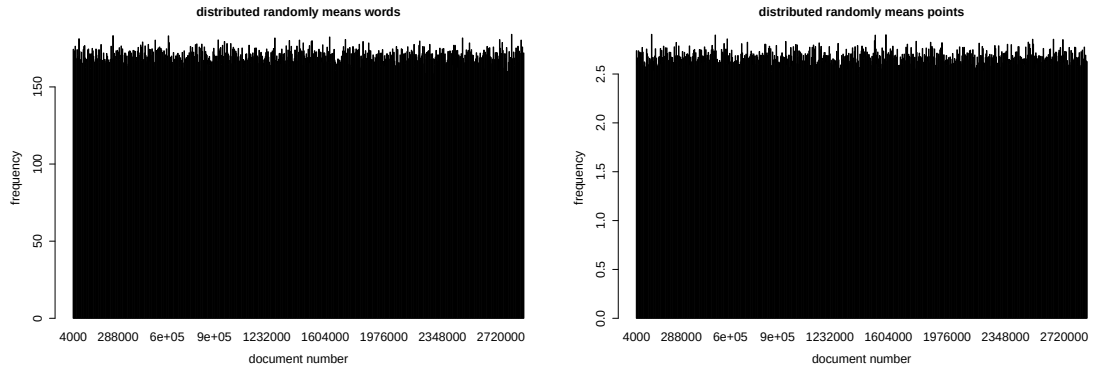
three phases. The first one is 'map', the second 'sort' and the third 'reduce'. The sort phase is responsible for assigning all elements with the same key to a list of values. Thus, an arbitrary user defined key is required which identically identifies the respective values. Initially, the article title is chosen as a key during the processing of the Wikipedia articles. As a post-processing step for the parsed articles, they are ordered by the article id. Regarding the Reuters documents, this does not seem to make an important difference as nearly all articles have a similar length or distribution of terms. The id assigned to the Reuters articles is allocated sequentially based on the event of their occurrence. Hence, the id correlates to the date or time when they are published.
This also applies to the Wikipedia articles. The page ids for the Wikipedia articles are assigned at creation time. Modifications can be seen in the form of revisions performed by users. Therefore, the system allocates sequentially generated ids based on the creation time of a page. The articles are sorted in an increasing order in figure 6.4. This figure also shows that articles having a lower id, which means that they are older, are longer than newer articles, at least with respect to the general tendency of an average value. An explanation for this could be given based on the modality of the work on a Wikipedia page. Generally, Wikipedia is a multiple authoring environment. If an article is created by one user, multiple others may help to correct, extend and review it. Changes may be introduced by everybody on such a page. Admittedly, there are some restrictions to provide references for facts or similar guidelines for editing. However, basically the page is created first and gets assigned the page id then which is not changed any more in the future. Only revision ids are kept to be able to reproduce the changes of the respective pages. Hence, older articles having a lower id number seem to have changed a lot over time and also treat very basic topics inside this encyclopedia. Thus, a larger number of individual terms is found inside these articles. Newer articles, however, have not changed so much up to the given time of inspection. This explains the values which can be seen in figure 6.4.
Still, the question is if a similar performance for the hybrid index can be achieved with the Wikipedia articles in use. Actually, at least regarding general tendency parameters from the statistical values (median and mean amount of terms and points), the two inspected corpora are very similar. Therefore, actions could be taken to provide the Wikipedia articles with a similar behaviour concerning the reorganisation runtime. That means that the articles could be inserted in a different order. Hence, a proper alignment is searched to fit the elements to a distribution which tries to resemble all portions in the total average of the number of terms. The first attempt towards this direction is to simply the ordering of the Wikipedia articles by their titles. The way of analysing the articles sorted by the titles is given in figure 6.5. It can be seen that the number of points inside the articles is closely related to the length of the text. Unfortunately, the distributions of the lengths of the articles does not match with the total average in some cases (total mean of 170 and median of 88). In some cases, e.g. near the end, the average text lengths is much greater $\approx 1,000$ terms per document and in some cases it is much smaller. Therefore, great differences in the runtime may again be expected. Another option for ordering the elements would be to reorganise the elements to fit to the expected distribution.



(a) Mean Words (b) Mean Points

Figure 6.5: Mean Values for Number of Terms and Number of Points (Wikipedia Dataset sorted by Article Titles)

Consequently, the articles should be re-ordered in a way that each of the particularly inspected part-collections of the articles fits to the given mean or median values. For this reason, a random selection model is applied which is supposed to pick the articles in a random order. The basic intention behind this approach is that if the random number generator is representative enough, a sufficiently random order of the articles is also generated and the mean (or median) values for the distributions fit to the desired output. Another option is to apply elaborate methods for fitting the distributions to a target model. However, similar results are likely to obtain when applying random generators with adequately precise metrics.



(a) Mean Words (b) Mean Points

Figure 6.6: Mean Values for Number of Terms and Number of Points (Wikipedia Dataset sorted randomly)

The results of the analysis of average numbers of terms and points for the randomly distributed Wikipedia dataset are displayed in figure 6.6. It depicts that these figures tend to show that there are only small deviations from the total mean of 170 for the randomly generated dataset. The point values are distributed similarly, too, so that each bucket of 4,000 elements resembles an internal mean value of 2.67. Therefore, it can be stated that the application of the random sorting approach to the document set resembles the target distribution very well. Smaller deviations from the target distributions exist but the result is already quite good.
For this reason, an additional test run, comparable to the initial ones is executed to verify the effects on the re-sorting of the articles.
The insertion times for the first 1,500 documents of the corpus are integrated into the database table where the index is constructed and the time is taken for each reorganisation run. The basic setup is identically equal to the one in the initial tests executed on the first, second and third

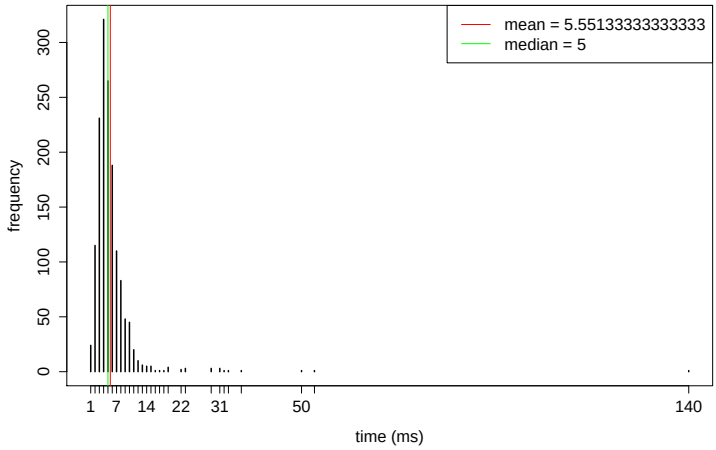


Figure 6.7: Wikipedia Randomised Order

portion of the Wikipedia corpus excerpt (sorted by the article id) and the test run on the Reuters corpus. The results shown in figure 6.7 are generated in the same way as the initial results. Multiple executions are also carried out to verify the final outcome and to stabilise the output. It is obvious that the average duration for one insertion operation of 5.513 ms is slightly lower than the one of the Reuters corpus (5.57 ms). However, this might also result from minor variations on the local machine.
Finally, it can be stated that the part of the Wikipedia which is worked on during the entire optimisation iterations seems to be the worst case of the test data. Articles having a higher page id are tendentially smaller regarding the number of terms per document as well as the amount of points. From a general point of view, the respectively inspected corpora do not differ much, at least with respect to parameters for a general trend like the mean and median. However, the order of the articles is important because local disturbances may influence the results. This is especially given for the first subsets of documents if they are arranged by their page id. Using a sorting approach which maps the entire collection to resemble the average values enhances the performance of the hybrid index in relation to this set of documents to a value which is very close to the one given in the Reuters collection.

6.3 EMPIRICAL DETERMINATION OF *BLENGTH*

In this work, there are two free parameters which are set arbitrarily during the experiments: *HLimit* and *BLength*. *HLimit* is a factor that separates the set of non-normalised or textual entries into two disjoint subsets of low and high frequently occurring values. A way to determine this parameter based on a minimisation is presented in section 4.2.4.4. This limit has major influences on reorganisation as well as on retrieval tasks. The second parameter, *BLength*, defines the length of the bitlist allocated per hybrid R-Tree element. It is also closely related to the corpus under test and must thus be determined for each corpus individually. For the experimental determination, the Reuters TRC2 as well as the Wikipedia corpus are taken like in the remaining parts of the work. The focus of this study is related to parameters for efficient retrieval functionalities. If there are more high frequent terms than the bitlist is able to store, additional R-Trees must be instantiated in order to persist all frequent terms. Therefore, an efficient query behaviour is expected if all query keywords can be found in not more than one hybrid R-Tree. The target of this investigation is to find a setting for the bitlist length for both corpora, provided a nearly optimal retrieval behaviour can be observed.

Note that the main results of this section can be found in [62]. All figures and tables are taken directly from this article.
A dataset of queries is required for assessing the optimal query efficiency. A set of queries is derived from the AOL Query Log[1]. The raw queries must be processed to derive meaningful ones which can be used to raise requests to the geo-textual datasets stored in the index. The original intention of the query set is to find data inside the Wikipedia dataset. Therefore, only queries whose results link to Wikipedia are kept and then, again, only those containing a place name (toponym) and a geographic region description, e.g. an administrative district. If the region is not given, the query region is constructed randomly by using an extent of 15° in each dimension, longitude and latitude. Next, the geographically relevant keywords are removed. The remaining keywords stay as they are supplied by the query log. Stemming and normalisation is also applied to the keywords to provide them in a similar way as the terms given in the corpora. A fixed document set is used as input to the hybrid access method. 200,000 document instances from both datasets are selected based on the ids. The Wikipedia dataset is ordered with respect to the article ids, whereas the Reuters articles get assigned an article id based on the publication date. In addition, the block size inside the dataset is set to 8192 bytes per page. A static set of input points is selected resulting from a gazetteer containing all point values which are likely to occur in any of the documents. The points are then used to apply a tree packing method to the R-Tree (see [37]) which guarantees logarithmic search time complexity for two-dimensional geographic objects. There are 14 different alternatives of bitlist lengths under test starting with 125 elements per bitlist. The amount of elements is doubled up to a value of 2,000 and increased by 2,000 up to a value of 20,000. The measurement runs use aspect oriented programming techniques to monitor page accesses during the retrieval of the results from the 699 queries originating from the query log analysis.

	Reuters							Wikipedia							
Frequ Terms	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	$\sum$
1	23	296						63	256						319
2	2	32	209					7	59	177					243
3		4	14	80				2	8	23	65				98
4				3	17				1	1	12	6			20
5			1		3	10			1	1	1	7	4		14
6						1	4						1	4	5
$\sum$	25	332	224	83	20	11	4	72	325	202	78	13	5	4	699

Table 6.2: Number of Keywords and Quantity of Affected Queries (adopted from [62])

The distribution of the queries with one to six keywords can be depicted from table 6.2. In addition, for each distinct amount of total keywords the table also presents the number of frequent ones. More than one (frequent) term is more relevant for this analysis because the occurrence of one single (frequent) keyword leads to the fact that not many pruning effects are expected during the traversal of the augmented R-Tree. Thus, as only one frequent keyword is given, a larger branching factor and consequently a smaller bitlist size is favoured, because only one R-Tree is required to be searched in any case. If the branching factor is high also not many levels inside the tree need to be visited leading to an optimal performance for small bitlist sizes. If the number of terms per bitlist rises, a smaller quantity of elements may be stored per node which causes a decrease in the branching factor and also more levels must be visited, which is disadvantageous for the retrieval efficiency. In order to limit the number of varying parameters for this study, the artificial upper bound *HLimit* is fixed to 300 for both corpora. The connection between bitlist length and R-Tree element count is shown in table 6.3. As already noticed, an increased number of elements per bitlist lowers the quantity of R-Tree elements per node and vice versa. That results from the fixed block size. If it is necessary to store more values inside

[1] `http://www.gregsadetsky.com/aol-data/`, accessed 2014-05-09

BLength	125	250	500	1000	2000	4000	6000
Elements	109	89	66	43	26	14	10
BLength	8000	10000	12000	14000	16000	18000	20000
Elements	7	6	5	4	3	3	3

Table 6.3: Bitlist Lengths and R-Tree Element Counts (adopted from [62])

the bitlists per element, the total required space on secondary memory increases, which leads to a lower fanout.
Table 6.3 demonstrates that there is no change of quantity of elements per R-Tree node between a bitlist length of 16000 and 20000. Hence, a saddle point probably exists in the subsequent analysis for these cases. Based on a manual evaluation, three different categories of

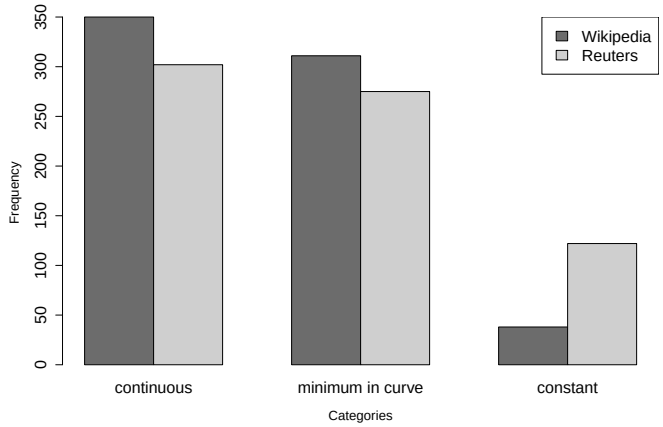
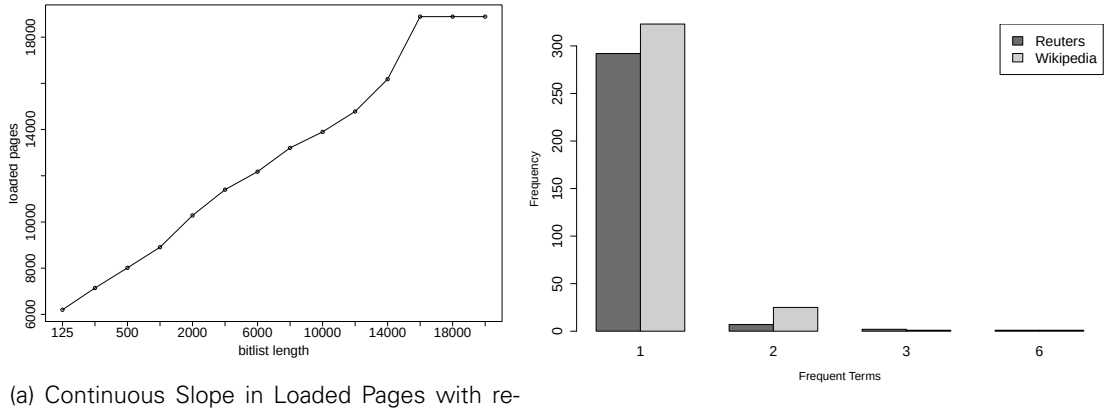


Figure 6.8: Frequency of Query Categories (adopted from [62])

the query behaviours arose: "*continuous*", "*minimum in curve*" and "*const*". The quantities of the respective cases can be depicted from figure 6.8.
The "const" case results from the fact that either none of the query keywords is present inside the index or only a fixed and low amount of references to this term is stored there. The basic behaviour leads to the constant case, because no R-Tree searches are issued, then. As mainly the R-Tree and the affected secondary inverted index structures are the places inside the structure to load the most pages and the initial inverted index as well as the document heap are loaded even when no (frequent) keyword is found, the results are nearly invariable. These facts lead to the constant case, because the alteration of the bitlist length only affects the retrieval efficiency in the hybrid R-Tree part which is untouched for terms with a low frequency.
The second category is the "*continuous*" behaviour.

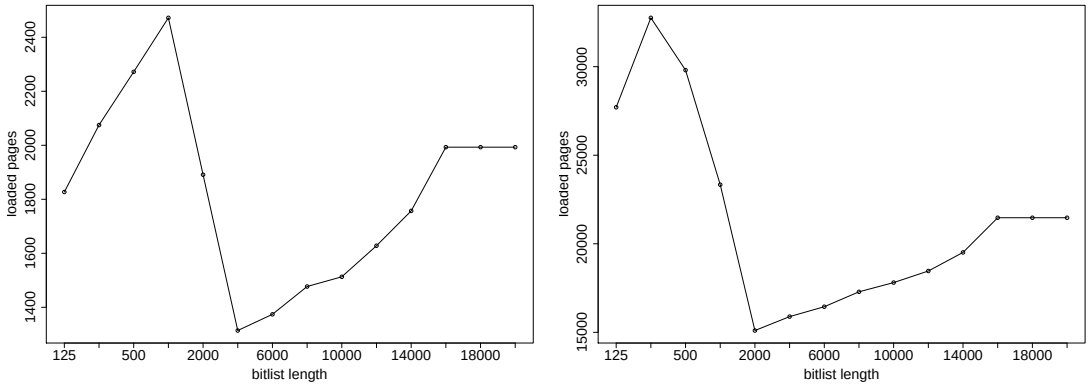
The instances of the "*continuous*" cases are shown in figure 6.9. One example of the resulting curve for one particular query is given in plot 6.9a. It shows the saddle point of a bitlist length between 16,000 and 18,000. Additionally, the total quantity of loaded nodes during one query continuously increases. The occurrence frequency of the "*continuous*" cases can be depicted from 6.9b. For both corpora the "*continuous*" behaviour emerges for one frequent term, in most of the cases. This can be explanined by the fact that a rise in the number of values to be stored inside one bitlist induces a larger amount of required storage space which lowers the numbers of elements per R-Tree node. This behaviour is listed in table 6.3. Thus, a larger number of levels and also a greater amount of nodes must be visited during a search request.
The most interesting behaviour in this study is given in the so-called "*minimum in curve*" cases. They are present in both inspected corpora. Typical instances of this behaviour are shown in figure 6.10 where one is listed per corpus. There is a monotonically increasing slope up to a



(a) Continuous Slope in Loaded Pages with respect to Bitlist Length

(b) Continuous Cases per Frequent Terms

Figure 6.9: Continuous Case (adopted from [62])



(a) Minimum Case (Reuters Dataset)

(b) Minimum Case (Wikipedia Dataset)

Figure 6.10: Examples for Minimum Case (adopted from [62])

certain bitlist length, followed by a maximum. From this point inside the curve, a monotonically decreasing manner occurs until a minimum is reached. Then, again, a monotonically increasing slope can be depicted. This results from the distribution of terms inside the bitlists. It must be noted that, in order to enable the storage of an arbitrary number of terms inside the bitlists, eventually multiple R-Tree instances are present, each of which stores a disjunct range of terms. If, for example, the bitlist size is limited to 250 and the 251st term needs to be stored inside the augmented R-Tree part, an additional R-Tree instance must be created which then stores terms with bit ids between 251 and 500. This explains the first part of the curve with a monotonic increase in the number of loaded nodes up to the maximum. In this case, multiple distinct R-Tree instances must be sought for the presence of the bits. The second section of the curve represents the decreasing slope. During this trend, a higher number of terms is found inside one bitlist. The minimum in the curve represents the shortest size of the bitlist where all frequent terms of the query can be found in one single R-Tree. This behaviour is followed by a section with monotonic increase which resembles exactly the same flow as in the "*continuous*" manner with a larger number of terms inside one bitlist leading to a smaller branching factor. The occurrences of the minimum values per bitlist length are demonstrated in figure 6.11. The distribution for both corpora is very similar to a normal distribution whereas the maximum frequencies are shifted. The Reuters dataset shows a maximum at a bitlist length of 4,000 Wikipedia, on the contrary, has a maximum at 2,000. The mean value for Wikipedia is 3,140 with a standard deviation of 2,585.207 and for Reuters 3,385 with a standard deviation of

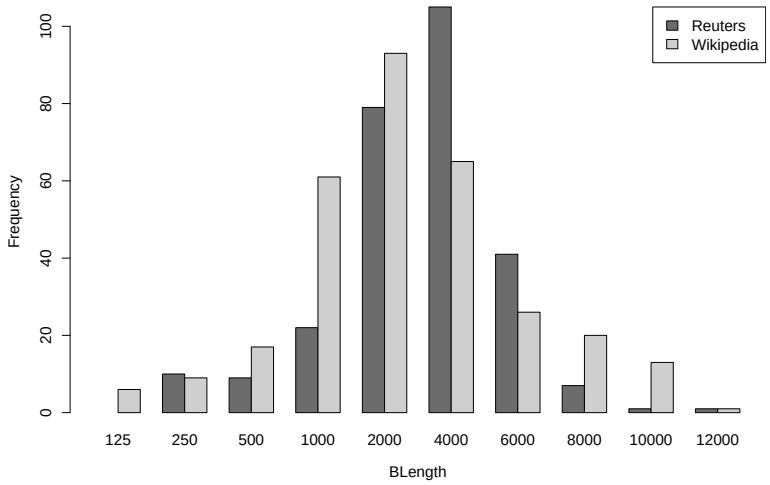


Figure 6.11: Frequencies of Minima for Both Datasets (adopted from [62])

1, 945.439, respectively. In total, an approximate bitlist length of 3, 000 seems to be a good choice for most of the cases.
It can be seen that if all high frequently occurring keywords are present in one R-Tree, the query efficiency solely depends on the branching factor given in the nodes. This also shows that a lower branching factor which leads to a bigger number of levels to be visited between the root and the leaf nodes is worse than the attempt to keep the quantity of levels small. Thus, the smallest bitlist length inducing the highest branching factor which contains all query keywords inside one single R-Tree is the optimum in all cases. Regarding the instance of one highly frequent term that occurs only at maximum, the smallest given bitlist length of 125 in this experimental evaluation showed the maximum performance.
The main results including the figures and tables of this study are already published in [62].

6.4 OVERALL TEST RESULTS FOR THE ENTIRE OPTIMISATION ITERATIONS

During the optimisation iterations, each state of the program is investigated individually. After one iteration, post-tests are executed checking the final outcome of the iteration against the initial state. Hence, only the comparisons of a pre- and a post-test of one particular iteration can be given as a temporary result.
The contents of this study including the figures and tables can be found in [63], too.
The entire optimisation process is also worthwhile investigating to get an overview of the actions performed and the outcome regarding the reorganisation procedures. For this reason, an empirical validation is set up to monitor the progress of the optimisation as an overview. An adopted version of the test suite which is used inside the optimisation iterations is applied to execute the test iterations. The identical set of documents serves as input for this study. That is why the quantity of documents has not changed at all for the optimisation iterations. The parameters do basically not change much. The only modification in the parameters is the application of the adopted split and choose subtree operation.
Due to time restrictions, only 38 repetitions of the test suite are executed for measurement reasons. This is mainly imposed by the inefficiency of the initial implementation. The amount of documents used in the inspected iterations is thus $n = \lceil \frac{38}{2} \rceil \cdot 50 + \lfloor \frac{38}{2} \rfloor = 969$.
The parameter settings for the test runs are described in table 6.4. They are the standard settings for the given iterations in the respective sections, too. The document count is set to

Parameter	Value	Explanation
HLimit	200	Artificial Upper Bound
Element Count	5	Amount of Elements per R-Tree node
Document Count	969	Amount of Documents to be inserted
Dataset	Wikipedia	Dataset to be processed (Wikipedia sorted by ids)
Split	R-Tree Split (Iteration 1–2) and R*-Tree Split (Iteration 3 – 7)	R-Tree splitting method
Choose Subtree	R-Tree Choose Subtree (Iteration 1–2) and R*-Tree Choose Subtree (Iteration 3 – 7)	Method for Selecting a proper subtree to place an element in

Table 6.4: Settings for the Test Suite Runs

969 based on the time limitations which are given owing to iteration 1. Since this is the particular optimisation recommendation for this iteration, the R-Tree split and choose subtree methods change in iteration 3. The remaining settings stay the same over all iterations. It is important to notice that the quantity of documents does not change at all because this study is executed to provide an overview of all optimizations and to summarise the results of this thesis as a consequence.
The particular iteration descriptions used here are slightly different from the ones described in this thesis. Iteration 1 is the initial state of the hybrid index before the start of the optimisation iterations. Hence, the phases are the following:

1. Initial State

2. Page and Value Caches (see section 5.2.2)

3. R-Tree Distribution (see section 5.2.3)

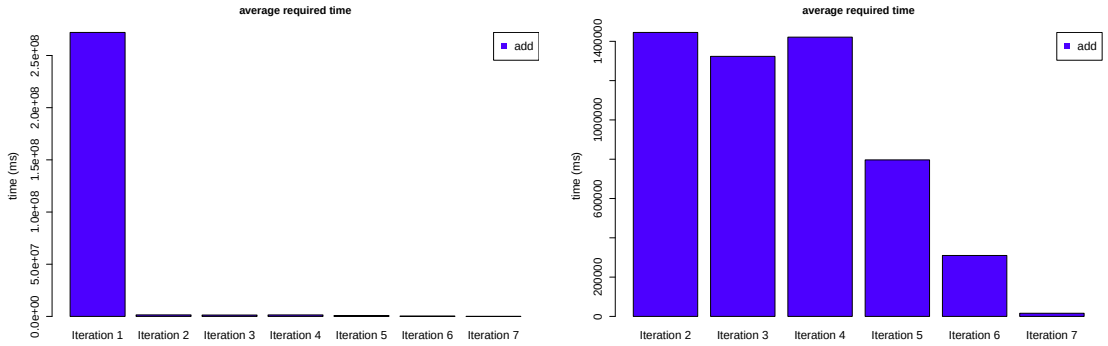
4. Inverted Index and Storage Changes (see section 5.2.4)

5. Advanced Inverted Index (see section 5.2.5)

6. Pre-Calculation of Item Insertions (see section 5.2.5)

7. Spatial Structures for Spatial Distributions (see section 5.2.6)

The changes in the iteration numbers assignment mainly result from the combination of multiple iterations in one description.
An overview of the average duration of the reorganisation runs is presented in figure 6.12. Two figures are given in this case because the differences between iteration 1 and 2 are simply too big to determine the exact process in one single graph. The second subfigure in this figure (see 6.12b) describes the trend of iterations 2 – 7. Inside the plots, a clear decrease of the runtime for each iteration, except the fourth, can be depicted. The left subgraph does not display much more than the fact that the runtime difference between iteration 1 and 2 is really tremendous. The remaining phases do not show comparable gaps. In contrary to the individual execution of tests which can be found in section 5.2.4, the average runtime rises for the fourth iteration. This is explainable based on the changes introduced in iteration 4. An improved way for the distribution of elements to the inverted index is introduced there, which includes a frequency dependent placement at different storage structures, either inside the leaf or in an additional page. The previous algorithm just places everything at a leaf node whereas an additional decision must be taken in the new version. Unfortunately, this only applies to larger document



(a) Overview of All Iterations

(b) Overview of Iterations 2 – 7

Figure 6.12: Overview of Average Time Required for Adding an Item (adopted from [63])

sets. Hence, a lot of documents must be inserted to the database until the distinctions between the storage strategy at the (secondary) inverted index introduces actual differences. The set of documents for the entire test runs, however, is only very limited (969). Because of the fact that the algorithms for inserting the elements to the secondary inverted index become more complex here, it is obvious that the average runtime also tends to be slightly higher based on the increased effort of the cases which must be handled for a low number of elements. Thus, this part of the algorithms only has major influences if the stored document set grows. As can be seen in section 5.2.4 where an input set of 1989 documents is used, the effects for bigger sets of documents become clearly recognisable.
The following subsections take the respective phases of the reorganisation algorithms into account and individually analyse the changes performed during the particular iterations. Based on the tremendous differences between iteration 1 and all the others, the first iteration is omitted in many of the subsequent illustrations.

6.4.1 Initial Inverted Index

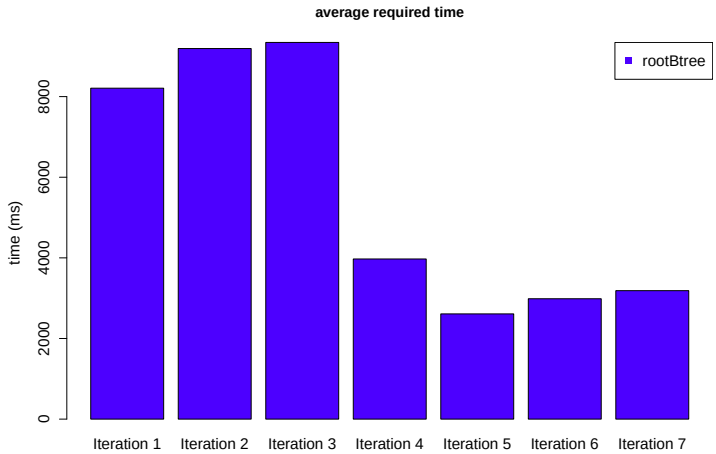


Figure 6.13: Overview of Initial Inverted Index Iterations

Figure 6.13 shows a summary of the iteration phases for the initial inverted index. Principally, as none of the iterations has identified this storage structure as the reason for the performance

issues, no optimisation is directly targeted towards this phase in the reorganisation of the hybrid index either. Therefore, the effects shown in the graph may be regarded as side effects of other optimisation iterations. Additionally, the measured times could be considered as very high, but they were taken with aspects enabled which introduces additional time per function call execution and thus distorts the results slightly. It can be seen that the average time required marginally rises between iteration 1 and 3. These iterations have basically introduced a cache which does not at all work on values of the initial inverted index or restructurings of the R-Tree. The most influential iterations for the manipulation of the initial inverted index are 4 and 5. Between 5 and 7 the average time required also increases slightly. Iteration 4 manipulates the inverted index storage by applying a hybrid storage strategy of the entries which separates between items with a high and a low frequency. Besides, also the storage of entries is manipulated in all storage structures by re-working the internal mechanisms. The introduction of improved mechanisms for finding a proper element inside one node by applying a bi-section oriented search approach shows results in the overview graphics (iteration 5) as well.
In total, the manipulation of the initial inverted index does not have much influence on the total runtime of the reorganisation algorithms of the hybrid index. Hence, no further inspection of the overview picture needs to be done.

6.4.2 R-Tree

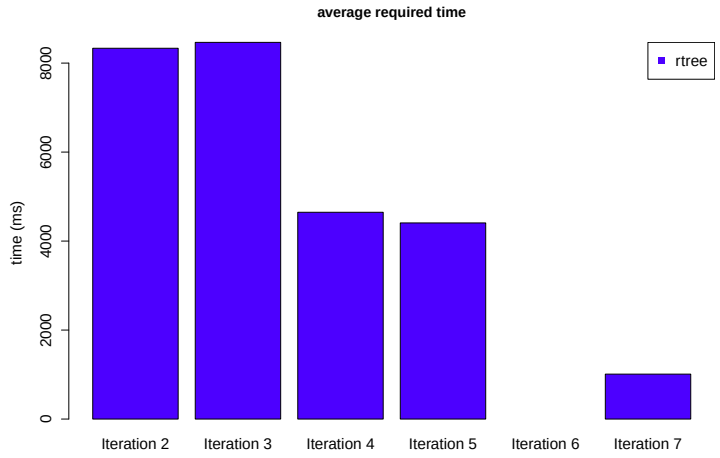


Figure 6.14: Overview of R-Tree Optimisation Iterations (adopted from [63])

The modification of the R-Tree operations is not subject of investigation during the optimisation iterations either. Basically, there are also some side-effects introduced by optimizations on other storage structures. A summary of the average modification runtimes for this phase can be depicted from figure 6.14. It displays the values for iterations 2 – 7. Iteration 1 is omitted here because the caching mechanism introduced in iteration 1 has significant effects on the runtime. The LRU value cache established in iteration 1 deals with the values stored in the R-Tree. The value-to-object-cache hides accesses to the spatial data and inner elements of the R-Tree. Hence, the influence of the first iteration is tremendous here.
Actually, iteration 3 provides a proper distribution of the elements through the R-Tree. It may be assumed that these changes also contribute to the runtime of the R-Tree algorithms. Unfortunately, the duration for this phase is slightly longer than in the previous phase. The actual calculation of the enhanced distribution requires a bigger amount of time. At least, the linear version of the R-Tree distribution and selection of a subtree is very straight-forward and thus also fast. However, it cannot provide an optimised distribution of the elements in the tree. The replacement algorithms of the R*-Tree are, from a computational perspective, by far more

complex. This involves the creation of different lists and the assignment to optimally distributed groups inside the nodes, which leads to a higher effort inside the algorithms. As only a very limited number of documents is used in the test set, the main effects are not as high as in the actual execution of the iteration.
Similar to the entire optimisation procedure, the R-Tree phase also shows the largest improvements between iterations 4 and 7. Iteration 6 is missing in this plot, because the R-Tree manipulation is included directly in the distribution phase for this optimisation stage. The difference between iteration 3 and 4 results from the adopted storage mechanism version. Beforehand, the data are required to be loaded from disk and then deserialized to one object which could thereafter be decomposed to its individual components. The new version includes the possibility of a more direct access to the individual storage parts. For this reason, the spatial components may directly be loaded from the page. The important change here is that the value cache is limited by the main memory. In iteration 3, the composed objects are memorised in the value cache. After that, only the actual spatial components (ranges or points) are used there. Based on the fact that the management of the spatial structures requires less main memory, more entries may be maintained in the cache. This leads to the increase of performance at this stage. As already mentioned, the value for iteration 6 is missing because of the internal change of the algorithm in this passage. There is, however, a large gap between iterations 5 and 7, which results from the introduction of the KD-Tree based distribution. The new algorithm pre-calculates the individual point values for one insertion run and assigns all textual occurrences to the respective points, directly. The R-Tree expansion step is only executed for points which are not present, yet. The first step in this algorithm is the distribution of points to the R-Tree. The points are removed from the KD-Tree, if they may be successfully deployed. That means that the remaining point occurrences are not yet included inside the final hybrid R-Tree. Hence, the insertion of the point values is delayed in this iteration and it is only executed for point values which are not present at all within the R-Tree. The old approach took each of the points into account, first performed a lookup for their presence and, if they did not exist, placed the elements to the R-Tree. Therefore, the relevant R-Tree elements had to be loaded (at minimum) twice. In addition, the KD-Tree already represents a spatial structure for main memory use which supports the separation and distribution of elements in a spatial manner. For this reason, it probably profits from caching effects during the traversal of the R-Tree, too. The KD-Tree already supports the spatial alignment of elements with a certain proximity. Hence, this spatial distribution is also beneficial for repetitive loading procedures of point values inside the hybrid R-Tree.

6.4.3 Secondary Inverted Index

The secondary inverted index manipulation is a very frequently used part of the algorithm and also a very important one. As it is the last step inside the algorithm, it is investigated repeatedly during the optimisation iterations. It also has direct impact on the distribution phase, because the two parts may not be analysed independently. Yet, it is presented before the actual description of the distribution phase, since it does not have any dependencies and may be inspected individually, whereas the distribution then directly relates to performance changes in the secondary inverted index manipulation.
A graphical overview of the iterations of the secondary inverted index manipulation phase can be depicted from 6.15. This figure excludes iteration 1, again, based on the enormous processing time differences. Due to the general behaviour of the access method, this phase is executed very frequently. Principally, for each point contained in at least one document which is referred to by at least one frequent term in the current reorganisation operation, the secondary inverted index must be manipulated. In addition, depending on the algorithm, for each term which cooccurs with the respective point the secondary inverted index is associated to, the phase is also performed once. This demonstrates that in case of a large runtime of one execution of this phase, the entire reorganisation procedure is decisively influenced. Based on the data shown in figure 6.15, the average runtime decreases steadily with exception of iteration 4. This seems to be confusing, because iteration 4 is targeted towards the optimisation of

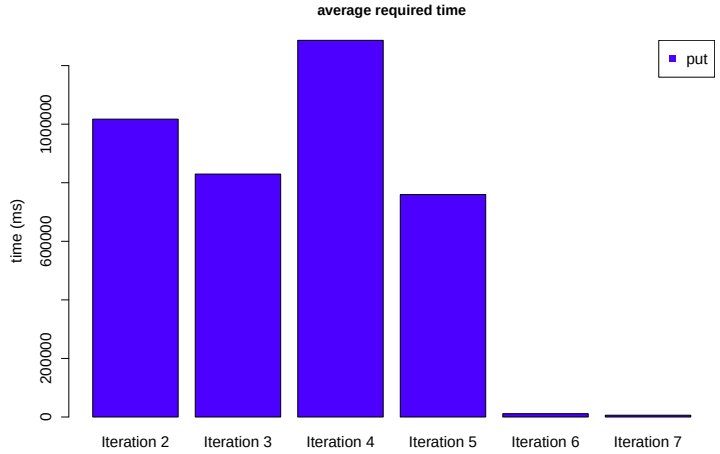


Figure 6.15: Overview of Secondary Inverted Index Manipulation Iterations (adopted from [63])

inverted index structures and storage mechanisms. Yet, it must be kept in mind that this phase still performs better in iteration 4 than in iteration 1. The change in the storage structures does not have any major influences on this part of the access method. Actually, the secondary inverted index simply maintains an assignment from a term id to a document heap reference. Thus, one integer value to a long assignment must be stored. The more direct way of accessing individual parts of composed objects does not influence it at all since the term id (as integer representation) is already stored in an integer value before and does not alter much.
The remaining changes introduced in iteration 4 refer to the placement of elements of the secondary inverted index. Three options are present, here. The newly introduced strategy called HYBRID refers to a frequency based differentiation of storage places. If the occurrence frequency of a particular term is greater than a pre-defined value, e.g. a block size, the actual values are moved or stored in an external inverted page. Otherwise, they are directly kept inside the directory B-Tree. The strategy before this change was to place all values to the directory. It is obvious that the determination of the proper place for storing an item is, computationally, more complicated for the new alternative. In addition, the management of the links between the respective page types is more complex in the new version. The HYBRID strategy also requires a certain frequency of values to be applied. Until this limit is reached, all values are placed to the secondary inverted index in the "old-fashioned" way. Thus, the strategy only applies to larger document sets. The set size of 969 documents under test is really small. Yet, it was required to use only a small document set based on the abysmal performance of the first iteration. In the actual measurement for this iteration, the document count is set to 1989, which is more than twice as large as the cardinality used by this study. In the respective measurement, the algorithmic changes are also very beneficial. However, due to the small number of documents in this study, the HYBRID distribution approach is nearly the same as the initial one (LEAF) but far more computationally complex, which leads to the runtime increase.
Except iteration 4, the remaining iterations show a clear decreasing tendency of the average runtimes required per execution. The adopted way of the distribution of point values seems to have a side-effect on the performance of the secondary inverted index manipulation. This can be explained owing to caching effects. If a lower number of R-Tree nodes must be loaded, the database page cache may be utilised for more secondary inverted index pages, which leads to a better throughput. However, with exception of the first iteration, the most striking changes for this phase occur between iteration 5 and 7. Iteration 6 includes the bulk-loading of secondary inverted index elements. This bulk-loading operation is executed per affected node and leads to the effect that a lower number of tree traversals need to be executed during the insertion. Hence, also a lower number of nodes must be loaded, because the new version of the algorithm also inspects sibling nodes for the possibility of placing additional elements, there. Besides, the treatment of points occurring in exactly one document effects a big profit in the runtime, as well.

The adoptions leading to the improved behaviour cannot be defined unambiguously, but their combination definitely has major influences. This phase also has the biggest consequences on the average required runtime per operation.
Iteration 7 is not targeted directly towards the reorganisation performance at secondary inverted index structures. Yet, based on the optimised distribution behaviour and a potentially better cache utilisation due to avoiding to load the entire affected R-Tree parts multiple times, this stage slightly improves the runtime, too. Still, the biggest differences can be depicted between iteration 5 and 7.

6.4.4 Distribution of Entries

Secondary inverted index and distribution of entries phases have turned out to be most influential for the entire reorganisation process during this thesis. The two individual phases have a tight connection to each other. In most cases, the `distribute` phase directly references the manipulation of secondary inverted index structures. This also leads to the the fact that changes in the final placement of term id to document references largely affect the performance of the executing and thus time-dependent phase.

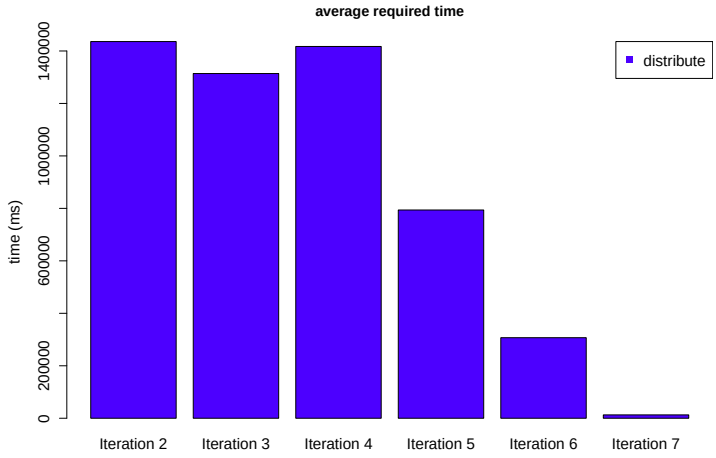


Figure 6.16: Overview of Distribution Optimisation Iterations (adopted from [63])

The evaluation of the distribution phase is displayed in figure 6.16. Similar to a lot of other phases, iteration 4 is also worse than iteration 3. In most iterations, the distribution is executed by traversing recursively through the hybrid R-Tree, generating subsets of valid elements and placing the final set at each of the secondary inverted index structures. Hence, the two phases of generating a set of elements applicable for the subtree pointed to by one element as well as the manipulation of the secondary inverted index directly influence the runtime of the distribution phase. The worsening of the runtime behaviour in iteration 4 is a direct conclusion of the influences of the secondary inverted index manipulation phase. Again, iteration 1 is omitted based on the tremendous differences. The changes in the distribution approach between iteration 2 and 3 are detectable, although they are relatively small. The fifth iteration introduces very large performance differences. It treats the handling of points occurring only in single documents separately. This seems to be have a decisive positive influence on the entire reorganisation process. Besides the fact of improving the reorganisation runtime, this procedure also leads to a better retrieval time, because no additional pages need to be loaded in case of the retrieval if at least one of these point values is included in the query region. Thus, no additional storage structure is affected either, if the retrieval may be stopped at the leaf node of a hybrid R-Tree whose element directly refers to the respective document. Iteration 6 introduces a relative performance increase of $\approx 50\%$ compared to iteration 5. This is mainly due to the

pre-calculation of the sets for the secondary inverted index structures inside a hash table. Apart from the hash map, the entire distribution algorithm is changed to a direct placement of the elements inside the leaf nodes. Whilst this is a successful adoption for this iteration, it is also the main time consumer. The hash table does not respect the spatial distribution, because the address inside the hash table is calculated on grounds of a combination of the values. Based on modulo operations, a potential spatial proximity of objects is destroyed. In the last iteration, the hash table is exchanged by a KD-Tree which provides a spatial treatment of the point values. Therefore, this structure has the possibility of keeping the spatial relationships between the points. By using this structure, the possibility arises to skip sublist generations, because it is sufficient if there is at least one point within the KD-Tree which lies inside the geographic region spanned by the node element. In addition, at the leaf node elements, the respective lists are already pre-calculated and ready for further operations. Therefore, no additional list generation needs to be executed. Due to the spatial alignment of the KD-Tree, it is also possible to execute queries for point values contained in ranges or exact ones, very fast. These effects lead to the substantial differences between iteration 6 and 7. Furthermore, the new distribution algorithm does not visit the affected R-Tree regions as often as the old one potentially does. This results in a better cache utilisation which affects other operations like the secondary inverted index manipulation as well.

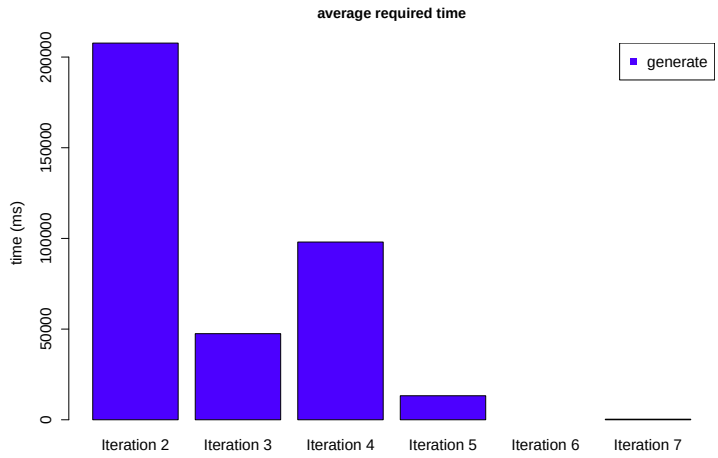


Figure 6.17: Overview of Sublist Generation (adopted from [63])

The progress of the runtimes of the particular iterations for the generation of the subsets is given in figure 6.17. Except for the outlier of iteration 4, the tendency is clearly a decreasing runtime for this phase as well. The evaluation of this algorithm phase shows a big decrease in runtime between iteration 2 and 3. Thus, it can be stated that it profits very much from the improved distribution of point values inside the R-Tree, which was also the desired effect for this iteration. The next notable change is introduced in iteration 7. Iteration 6 skips the generation of sublists, because the values are already prepared inside a hash map and thus, no list generation is necessary. Yet, iteration 7 pre-calculates all values required for the traversal of the R-Tree in advance and only uses the generation phase as lookup cache inside the KD-Tree. The KD-Tree is pre-calculated from the original data and thus may also be adopted for efficient retrieval. Therefore, the range queries which are issued towards the KD-Tree may be answered very fast. In addition, it is sufficient to know that points fulfilling the containment inside a particular range exist, since this is the only criterion at this stage. When the leaf level is reached, the pre-calculated sets for the point values may also be taken directly and placed at the secondary inverted index structures. Hence, the pre-calculation introduced great benefits for this phase as well.

This section provided an overview of the optimisation iterations. The input set as well as the outer parameters are fixed in the beginning. This leads to the use of exactly the same input set

and test circumstances for each iteration execution, which makes the respective values comparable. The phases are analysed regarding the improvements introduced for each iteration with respect to the entire process. Consequently, this section outlines the total results of the entire thesis. They are summarised in [63], which gives an overview of the effort done in this thesis.

7 CONCLUSIONS AND FUTURE PROSPECTS

7.1 CONCLUSION

This thesis describes the optimisation of the reorganisation algorithms of hybrid index structures, access methods that may be used for a large variety of application domains like enterprise content management or geographic information retrieval systems. In prior to this work, the access methods are primarily tested in simulation based systems. Unfortunately, they do not exactly reflect genuine conditions in realistic environments. A lot of effects can only be shown if the access methods are implemented in a database management system. Besides, little was known about the theoretical features of hybrid index structures in real world systems. The behaviour of the hybrid index structures regarding retrieval operations is already established as efficiently based on theoretical as well as empirical studies. However, little information was given about the proper reorganisation with a special focus on the insertion algorithms. Thus, the main target of this thesis is the application of innovative technologies and algorithm enhancements to make fast reorganisations possible, too. At the beginning of this thesis, the time frame required for the insertion operations was unacceptable for the setup of hybrid index structures in realistic environments like information systems. Generally, these information systems must especially be able to handle big datasets if they are embedded, e.g., in web based search engines. Therefore, fast restructurings must be made possible for this task.
This work first introduces a an initial implementation of the hybrid access method embedded in a realistic database system. As a next step, a theoretical model to derive the general properties and coherences of hybrid index structures in realistic database environments, a framework that may then be used to model hybrid index structures and track changes or implement adaptations inside the algorithms which refer back to the model is established. The next target of investigation is to find inefficiencies which must be overcome.
An optimisation framework is established comprised by corpora to be used as input sets, a test as well as an analysis framework which is applied for the optimisation iterations. The target of the entire work is to reduce the average reorganisation runtime to a reasonable duration. Typical insertion operations on currently used index access methods like the B-Tree only require several milliseconds. Thus, a goal of the optimizations is to limit the duration of an average restructuring to a millisecond time frame. Initially, a lot of insertions took several seconds to be completed. Finally, this time could be reduced to milliseconds. This performance gain is achieved by the application of the test suite which is specified in the beginning as well as by applying optimizations to paths inside the software suggested by an optimisation recommendation framework. The optimisation alternatives are either taken from literature or by applying

customised solutions based on additional studies targeted towards features that are extracted as not running well at the respective iteration.
Eventually, the algorithms in the final state are described as well as an overview is given of the entire iterations. Besides, also the differences between the respective parts of the corpora are outlined because it has been noticed that some portions of the Wikipedia corpus may be handled better than others. Thus, the entire optimisation is executed on the worst case of the Wikipedia portions which are, in total, also the worst case of the inspected document corpora.
This thesis answers the research questions given in section 1.2.1. As a first step, a general model is deduced for the use in database oriented application domains. Furthermore, a structured approach is implemented to derive the malicious parts of the algorithms from the hybrid index structure. A customised profiling based software optimisation recommendation and path pruning system is introduced in this step. The most important question in this case is to find options for optimizations of the hybrid index management algorithms without contributing negatively to the search time complexity. In general, due to some optimizations, the search time could even be improved whilst optimising the reorganisation algorithms.

7.2 FUTURE PROSPECTS

Based on a previous work and the results of this thesis, efficient retrieval as well as the reorganisation of hybrid index structures supporting multimedia search criteria are explored in detail. Yet, some open issues still exist which may lead to further research works.
One pending point of this research work is the application of ranking for this kind of data. Currently, solely a query class using boolean non-normalised and normalised results is included. That means that the only criterion is that a non-normalised value is comprised inside the set of non-normalised values for one tuple as well as a point or range description for the normalised dimension. This might, e.g., be a term in a set of keywords contained inside a text in conjunction with a point inside a given query range for geographic features. At the current state no ordering is applied to the results. Textual or non-normalised features could be ranked by using vector space model based cosine distances [89] or also standard tf-idf ranks [91]. These are some of the standard approaches employed for textual ranking which may also be applied to set based representations of values. Regarding the normalised dimension, other kinds of ranking order need to be investigated. In this case, multiple solutions could be possible. An ordering that results from a nearest neighbour oriented search option is imaginable. Besides, if multiple normalised values are allowed as well, also a measurement scheme using counts of objects inside and outside the query region could be investigated. Finally, the outcome should consist of a weighted sum of the partial weights. Yet, the respective schemes must be constructed and evaluated. This could include additional storage overhead leading to changes in the affected storage structures. Unfortunately, this is not the target of the current research work and must be solved in the future.
Another prospect could be the integration of this new kind of access structures into other database types. At the moment, only relational database management systems are being inspected. The implementations work on the PostgreSQL database as initial implementation followed by a port to the H2 database as it is described in this thesis. In the last years, many specialised database types have been developed like graph and in memory databases as well as key value or column stores. Each of these database systems supports specialised and efficient access to the respectively supported data types. Owing to the specific requirements of applications, the classical relational databases reached the limits, e.g. in social network based application domains. Therefore, these new kinds of NoSQL or NewSQL databases have been developed to support efficient retrieval of the specific data types. The integration of hybrid index structures into selected databases could also be a perspective for this technique. Probably, the integration in disk oriented databases may serve as a first step because the existing algorithms are optimised towards the use in disk based systems. For this reason, for example, graph database support for hybrid index structures seams to be feasible. Regarding other kinds of databases like in memory stores, the entire algorithms and storage structure layouts must be changed to provide sufficient support for the use there. In general it may be stated that this work and its predecessor build the basis for the implementation of hybrid index structures in hard disk oriented systems providing data structures and algorithms for efficient retrieval and the reorganisation of hybrid index structures supporting multimedia search criteria.

BIBLIOGRAPHY

[1] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation, PLDI '90, pages 246–256, New York, NY, USA, 1990. ACM.

[2] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. Data page layouts for relational databases on deep memory hierarchies. The VLDB Journal, 11(3):198–215, November 2002.

[3] Chuan-Heng Ang and T. C. Tan. 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, 1997. Springer-Verlag.

[4] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. Acta Informatica, 1:173 – 189, 1972. 10.1007/BF00288683.

[5] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. SIGMOD Rec., 19(2):322–331, 1990.

[6] Norbert Beckmann and Bernhard Seeger. A revised r*-tree in comparison with related index structures. In Proceedings of the 35th SIGMOD international conference on Management of data, SIGMOD '09, pages 799–812, New York, NY, USA, 2009. ACM.

[7] L.A. Belady. A study of replacement algorithms for a virtual-storage computer. IBM Systems Journal, 5(2):78–101, 1966.

[8] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. Commun. ACM, 18(9):509–517, September 1975.

[9] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency controltheory and algorithms. ACM Trans. Database Syst., 8:465–483, December 1983.

[10] Haran Boral and David J. DeWitt. Design considerations for data-flow database machines. In Proceedings of the 1980 ACM SIGMOD International Conference on Management of Data, SIGMOD '80, pages 94–104, New York, NY, USA, 1980. ACM.

[11] Eric W. Brown, James P. Callan, and W. Bruce Croft. Fast incremental indexing for full-text information retrieval. In Proceedings of the 20th International Conference on Very Large Data Bases, VLDB '94, pages 192–202, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[12] Stefan Büttcher and Charles L. A. Clarke. A hybrid approach to index maintenance in dynamic text retrieval systems. In In ECIR 2006: Proceedings of the 28th European Conference on Information Retrieval, pages 229–240, 2006.

[13] Ariel Cary, Ouri Wolfson, and Naphtali Rishe. Efficient and scalable method for processing top-k spatial boolean queries. In Michael Gertz and Bertram Ludäscher, editors, Scientific and Statistical Database Management, volume 6187 of Lecture Notes in Computer Science, pages 87–95. Springer Berlin Heidelberg, 2010.

[14] Chee Yong Chan, Beng Chin Ooi, and Hongjun Lu. Extensible buffer management of indexes. In Proceedings of the 18th International Conference on Very Large Data Bases, VLDB '92, pages 444–454, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[15] Lisi Chen, Gao Cong, and Xin Cao. An efficient query indexing mechanism for filtering geo-textual data. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD '13, pages 749–760, New York, NY, USA, 2013. ACM.

[16] Lisi Chen, Gao Cong, Christian S. Jensen, and Dingming Wu. 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, 2013.

[17] Yen-Yu Chen, Torsten Suel, and Alexander Markowetz. Efficient query processing in geographic web search engines. In Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, SIGMOD '06, pages 277–288, New York, NY, USA, 2006. ACM.

[18] Maria Christoforaki, Jinru He, Constantinos Dimopoulos, Alexander Markowetz, and Torsten Suel. Text vs. space: Efficient geo-search query processing. In Proceedings of the 20th ACM International Conference on Information and Knowledge Management, CIKM '11, pages 423–432, New York, NY, USA, 2011. ACM.

[19] E. F. Codd. A relational model of data for large shared data banks. Commun. ACM, 13(6):377–387, June 1970.

[20] Gao Cong, Christian S. Jensen, and Dingming Wu. Efficient retrieval of the top-k most relevant spatial web objects. Proc. VLDB Endow., 2(1):337–348, August 2009.

[21] D. Cutting and J. Pedersen. Optimization for dynamic inverted index maintenance. In Proceedings of the 13th annual international ACM SIGIR conference on Research and development in information retrieval, SIGIR '90, pages 405–411, New York, NY, USA, 1990. ACM.

[22] Michael Dunlavey. Performance tuning with instruction-level cost derived from call-stack sampling. SIGPLAN Not., 42(8):4–8, August 2007.

[23] Michael R. Dunlavey. Building Better Applications: A Theory of Efficient Software Development. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994.

[24] Wolfgang Effelsberg and Theo Haerder. Principles of database buffer management. ACM Trans. Database Syst., 9(4):560–595, December 1984.

[25] Klaus Elhardt and Rudolf Bayer. A database cache for high performance and fast restart in database systems. ACM Trans. Database Syst., 9(4):503–525, December 1984.

[26] Shanti Elizabeth, Kirutthika Raja, and Nadarajan Rathanaswamy. An indexing method for handling queries on set-valued attributes in object-oriented databases. In Stefan Edlich and James H. Paterson, editors, Object Databases, First International Conference, ICOODB 2008, Berlin, Germany, March 13-14, 2008. Proceedings, pages 143–160. Tribun EU, 2008.

[27] Robert Epstein and Paula Hawthorn. Design decisions for the intelligent database machine. In Proceedings of the May 19-22, 1980, National Computer Conference, AFIPS '80, pages 237–241, New York, NY, USA, 1980. ACM.

[28] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In PODS '01: Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pages 102–113, New York, NY, USA, 2001. ACM.

[29] Chris Faloutsos and Stavros Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. ACM Trans. Inf. Syst., 2(4):267–288, October 1984.

[30] Ian De Felipe, Vagelis Hristidis, and Naphtali Rishe. Keyword search on spatial databases. International Conference on Data Engineering, 0:656–665, 2008.

[31] Nathan Froyd, John Mellor-Crummey, and Rob Fowler. Low-overhead call path profiling of unmodified, optimized code. In Proceedings of the 19th annual international conference on Supercomputing, ICS '05, pages 81–90, New York, NY, USA, 2005. ACM.

[32] Leo Galambos. Dynamic inverted index maintenance.

[33] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In Proceedings of the 1982 SIGPLAN symposium on Compiler construction, SIGPLAN '82, pages 120–126, New York, NY, USA, 1982. ACM.

[34] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. SIGPLAN Not., 39(4):49–57, April 2004.

[35] Diane Greene. An implementation and performance analysis of spatial data access methods. In Proceedings of the Fifth International Conference on Data Engineering, pages 606–615, Washington, DC, USA, 1989. IEEE Computer Society.

[36] Antonin Guttman. 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, 1984. ACM.

[37] Richard Göbel. Towards logarithmic search time complexity for r-trees. In Tarek Sobh, editor, Innovations and Advanced Techniques in Computer and Information Sciences and Engineering, pages 201–206. Springer Netherlands, 2007.

[38] Richard Göbel and Antonio de la Cruz. Computer science challenges for retrieving security related information from the internet. Global Monitoring for Security and Stability (GMOSS), -:90 – 101, 2007.

[39] Richard Göbel, Andreas Henrich, Raik Niemann, and Daniel Blank. 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, 2009. ACM.

[40] Richard Göbel and Carsten Kropf. Towards hybrid index structures for multi-media search criteria. In Proceedings of the 16th International Conference on Distributed Multimedia Systems, DMS 2010, October 14-16, 2010, Hyatt Lodge at McDonald's Campus, Oak Brook, Illinois, USA, pages 143–148. Knowledge Systems Institute, 2010.

[41] Richard Göbel, Carsten Kropf, and Sven Müller. Efficiency of hybrid index structures - theoretical analysis and a practical application. In Erland Jungert, editor, The 20th International Conference on Distributed Multimedia Systems: Research papers on distributed multimedia systems, distance education technologies and visual languages and computing, Pittsburgh, PA, USA, August 27-29, 2014., pages 182–188. Knowledge Systems Institute Graduate School, 2014.

[42] Richard Göbel, Carsten Kropf, and Sven Müller. Efficiency of hybrid index structures - theoretical analysis and a practical application. J. Vis. Lang. Comput., 25(6):800–807, 2014.

[43] Ramaswamy Hariharan, Bijit Hore, Chen Li, and Sharad Mehrotra. Processing spatial-keyword (sk) queries in geographic information retrieval (gir) systems. In SSDBM '07: Proceedings of the 19th International Conference on Scientific and Statistical Database Management, page 16, Washington, DC, USA, 2007. IEEE Computer Society.

[44] Harold Stanley Heaps. Information Retrieval: Computational and Theoretical Aspects. Academic Press, 1978.

[45] Joseph M. Hellerstein, Elias Koutsoupias, Daniel P. Miranker, Christos H. Papadimitriou, and Vasilis Samoladas. On a model of indexability and its bounds for range queries. J. ACM, 49(1):35–55, January 2002.

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

[47] Alan Hevner. Design research in information systems theory and practice. Springer, New York London, 2010.

[48] Alan R Hevner. A three cycle view of design science research. Scandinavian journal of information systems, 19(2):4, 2007.

[49] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. MIS Q., 28(1):75–105, March 2004.

[50] David Hilbert. Ueber die stetige abbildung einer linie auf ein flächenstück. Mathematische Annalen, 38:459–460, 1891.

[51] Juhani Iivari. A paradigmatic analysis of information systems as a design science. Scandinavian Journal of Information Systems, 19(2):39 – 64, 2007.

[52] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. ACM Comput. Surv., 40(4):1–58, 2008.

[53] Christian S. Jensen. Spatial keyword querying of geo-tagged web content. In Proceedings of the 7th International Workshop on Ranking in Databases, DBRank '13, pages 1:1–1:4, New York, NY, USA, 2013. ACM.

[54] Thorsten Joachims. A probabilistic analysis of the rocchio algorithm with tfidf for text categorization. In Proceedings of the Fourteenth International Conference on Machine Learning, ICML '97, pages 143–151, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[55] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In Proceedings of the 20th International Conference on Very Large Data Bases, VLDB '94, pages 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[56] Christopher B. Jones, Alia I. Abdelmoty, David Finch, and Gaihua Fu. The spirit spatial search engine: Architecture, ontologies and spatial indexing. In In Proc. 3rd Int. Conf. on Geographic Information Science, pages 125–139, 2004.

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

[58] K. V. Ravi Kanth and Ambuj K. Singh. Optimal dynamic range searching in non-replicating index structures. In In Proc. International Conference on Database Theory, LNCS 1540, pages 257–276, 1997.

[59] Ali Khodaei, Cyrus Shahabi, and Chen Li. Hybrid indexing and seamless ranking of spatial and textual features of web documents. In PabloGarcía Bringas, Abdelkader Hameurlain, and Gerald Quirchmayr, editors, Database and Expert Systems Applications, volume 6261 of Lecture Notes in Computer Science, pages 450–466. Springer Berlin Heidelberg, 2010.

[60] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, ECOOP'97 — Object-Oriented Programming, volume 1241 of Lecture Notes in Computer Science, pages 220–242. Springer Berlin Heidelberg, 1997.

[61] Donald E. Knuth. The art of computer programming, volume 3: (2nd ed.) sorting and searching. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[62] Carsten Kropf. Determining parameters for efficient retrieval in index structures for hybrid data spaces. In Digital Information Management (ICDIM), 2014 Ninth International Conference on, pages 98–103, Sept 2014.

[63] Carsten Kropf. Towards efficient reorganisation algorithms of hybrid index structures supporting multimedia search conditions. In Markus Helfert, Andreas Holzinger, Orlando Belo, and Chiara Francalanci, editors, DATA 2014 - Proceedings of 3rd International Conference on Data Management Technologies and Applications, Vienna, Austria, 29-31 August, 2014, pages 231–242. SciTePress, 2014.

[64] Carsten Kropf, Shamim Ahmmed, Richard Göbel, and Raik Niemann. A geo-textual search engine approach assisting disaster recovery, crisis management and early warning systems. In Geo-information for Disaster management (Gi4DM), 2011.

[65] Carsten Kropf and Bertram Schlecht. Nerseng: Query analysis and indexing. In Learning, Knowledge, Adaptation (LWA), pages 52 – 58, 2013.

[66] Per-Ake Larson. Dynamic hash tables. Commun. ACM, 31(4):446–457, April 1988.

[67] James R. Larus. Whole program paths. In Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation, PLDI '99, pages 259–269, New York, NY, USA, 1999. ACM.

[68] Nicholas Lester, Alistair Moffat, and Justin Zobel. Fast on-line index construction by geometric partitioning. In Proceedings of the 14th ACM international conference on Information and knowledge management, CIKM '05, pages 776–783, New York, NY, USA, 2005. ACM.

[69] S.T. Leutenegger, M.A. Lopez, and J. Edgington. Str: a simple and efficient algorithm for r-tree packing. In Data Engineering, 1997. Proceedings. 13th International Conference on, pages 497 –506, apr 1997.

[70] Zhisheng Li, K.C.K. Lee, Baihua Zheng, Wang-Chien Lee, Dik Lun Lee, and Xufa Wang. Ir-tree: An efficient index for geographic document search. Knowledge and Data Engineering, IEEE Transactions on, 23(4):585–599, April 2011.

[71] Lipyeow Lim, Min Wang, Sriram Padmanabhan, Jeffrey Scott Vitter, and Ramesh Agarwal. Dynamic maintenance of web indexes using landmarks. In Proceedings of the 12th international conference on World Wide Web, WWW '03, pages 102–111, New York, NY, USA, 2003. ACM.

[72] Xing LIN, Bo YU, and Yifang Ban. On indexing mechanism in geographical information retrieval system. -, -:–, 2007.

[73] Linyuan Lü, Zi-Ke Zhang, and Tao Zhou. Zipf's law leads to heaps' law: Analyzing their relation in finite-size systems. PLoS ONE, 5(12):e14139, 12 2010.

[74] Silvano Maffeis. Cache management algorithms for flexible filesystems. ACM SIGMETRICS Performance Evaluation Review, 21:1–3, 1993.

[75] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. An Introduction to Information Retrieval. Cambridge University Press, 2009.

[76] Salvatore T. March and Gerald F. Smith. Design and natural science research on information technology. Decis. Support Syst., 15(4):251–266, December 1995.

[77] Donald R. Morrison. Patricia-practical algorithm to retrieve information coded in alphanumeric. J. ACM, 15(4):514–534, 10 1968.

[78] G.M. Morton. A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing. International Business Machines Company, 1966.

[79] J. Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. ACM Trans. Database Syst., 9(1):38–71, March 1984.

[80] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 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, 1993. ACM.

[81] M. F. Porter. An algorithm for suffix stripping. In Karen Sparck Jones and Peter Willett, editors, Readings in information retrieval, pages 313–316. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[82] Raghu Ramakrishnan and Johannes Gehrke. Database Management Systems. McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003.

[83] Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhardt, and Rudolf Bayer. Integrating the ub-tree into a database system kernel. In Proceedings of the 26th International Conference on Very Large Data Bases, VLDB '00, pages 263–272, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[84] David P. Reed. NAMING AND SYNCHRONIZATION IN A DECENTRALIZED COMPUTER SYSTEM. PhD thesis, Massachusetts Institute of Technology, 1978.

[85] João B. Rocha-Junior and Kjetil Nørvåg. 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, 2012. ACM.

[86] João B. Rocha-Junior, Orestis Gkorgkas, Simon Jonassen, and Kjetil Nørvåg. 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, 2011.

[87] W. W. Royce. Managing the development of large software systems: concepts and techniques. In Proceedings of the 9th international conference on Software Engineering, ICSE '87, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[88] Giovanni Maria Sacco and Mario Schkolnick. Buffer management in relational database systems. ACM Transactions on Database Systems, 11:473–498, 1986.

[89] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. Commun. ACM, 18(11):613–620, November 1975.

[90] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. Inf. Process. Manage., 24(5):513–523, August 1988.

[91] Gerard Salton and Michael J. McGill. Introduction to Modern Information Retrieval. McGraw-Hill, Inc., New York, NY, USA, 1986.

[92] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In Proceedings of the 13th International Conference on Very Large Data Bases, VLDB '87, pages 507–518, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[93] Herbert Simon. The sciences of the artificial. MIT Press, Cambridge, Mass, 1996.

[94] SPIRIT. Spatially-aware information retrieval on the internet. `http://www.geo-spirit.com/`, 10 2014.

[95] J. M. Spivey. Fast, accurate call graph profiling. Softw. Pract. Exper., 34(3):249–264, March 2004.

[96] Andrew S. Tanenbaum and Albert S. Woodhull. Operating Systems Design and Implementation. Prentice Hall of India, 2006.

[97] The Neo4j Team. The neo4j manual v1.9.2. `http://docs.neo4j.org/pdf/neo4j-manual-milestone.pdf`, 07 2013.

[98] Subodh Vaid, Christopher B. Jones, Hideo Joho, and Mark Sanderson. 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, 2005.

[99] Oscar Waddell and J. Michael Ashley. Visualizing the performance of higher-order programs. In Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, PASTE '98, pages 75–82, New York, NY, USA, 1998. ACM.

[100] Roel Wieringa. Design science as nested problem solving. In Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology, DESRIST '09, pages 8:1–8:12, New York, NY, USA, 2009. ACM.

[101] Dingming Wu, Man Lung Yiu, Gao Cong, and Christian S. Jensen. Joint top-k spatial keyword query processing. Knowledge and Data Engineering, IEEE Transactions on, 24(10):1889 –1903, oct. 2012.

[102] Dingming Wu, Man Lung Yiu, and Christian S. Jensen. Moving spatial keyword queries: Formulation, methods, and analysis. ACM Trans. Database Syst., 38(1):7:1–7:47, April 2013.

[103] Jianliang Xu, B. Zheng, W.-C. Lee, and Dik Lun Lee. The d-tree: an index structure for planar point queries in location-based wireless services. Knowledge and Data Engineering, IEEE Transactions on, 16(12):1526 – 1542, dec. 2004.

[104] Dongxiang Zhang, Yeow Meng Chee, Anirban Mondal, Anthony K. H. Tung, and Masaru Kitsuregawa. Keyword search in spatial databases: Towards searching by document. Data Engineering, International Conference on, 0:688–699, 2009.

[105] Yinghua Zhou, Xing Xie, Chuang Wang, Yuchang Gong, and Wei-Ying Ma. 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, 2005. ACM.

[106] George Kingsley Zipf. Human Behaviour and the Principle of Least Effort: an Introduction to Human Ecology. Addison-Wesley, 1949.

[107] Justin Zobel and Alistair Moffat. Inverted files for text search engines. ACM Computing Surveys, 38:2006, 2006.

[108] Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. Inverted files versus signature files for text indexing. ACM Trans. Database Syst., 23(4):453–490, December 1998.

CONTRIBUTIONS FROM THE AUTHOR

Richard Göbel and Carsten Kropf. Towards hybrid index structures for multi-media search criteria. In Proceedings of the 16th International Conference on Distributed Multimedia Systems, DMS 2010, October 14-16, 2010, Hyatt Lodge at McDonald's Campus, Oak Brook, Illinois, USA, pages 143–148. Knowledge Systems Institute, 2010.

Richard Göbel, Carsten Kropf, and Sven Müller. Efficiency of hybrid index structures - theoretical analysis and a practical application. In Erland Jungert, editor, The 20th International Conference on Distributed Multimedia Systems: Research papers on distributed multimedia systems, distance education technologies and visual languages and computing, Pittsburgh, PA, USA, August 27-29, 2014., pages 182–188. Knowledge Systems Institute Graduate School, 2014.

Richard Göbel, Carsten Kropf, and Sven Müller. Efficiency of hybrid index structures - theoretical analysis and a practical application. J. Vis. Lang. Comput., 25(6):800–807, 2014.

Carsten Kropf, Shamim Ahmmed, Richard Göbel, and Raik Niemann. A geo-textual search engine approach assisting disaster recovery, crisis management and early warning systems. In Geo-information for Disaster management (Gi4DM), 2011.

Carsten Kropf. Determining parameters for efficient retrieval in index structures for hybrid data spaces. In Digital Information Management (ICDIM), 2014 Ninth International Conference on, pages 98–103, Sept 2014.

Carsten Kropf. Towards efficient reorganisation algorithms of hybrid index structures supporting multimedia search conditions. In Markus Helfert, Andreas Holzinger, Orlando Belo, and Chiara Francalanci, editors, DATA 2014 - Proceedings of 3rd International Conference on Data Management Technologies and Applications, Vienna, Austria, 29-31 August, 2014, pages 231–242. SciTePress, 2014.

Carsten Kropf and Bertram Schlecht. Nerseng: Query analysis and indexing. In Learning, Knowledge, Adaptation (LWA), pages 52 – 58, 2013.

APPENDIX

A XML FILES

A.1 Test Suite XML Trace

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.iisys.de/trace"
        xmlns:trace="http://www.iisys.de/trace"
        elementFormDefault="qualified">
        <xsd:element name="trace" type="trace:call_list_type"/>
        <xsd:complexType name="call_list_type">
                <xsd:sequence>
                        <xsd:element name="call" type="trace:call_type"
                                minOccurs="1" maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="call_type">
                <xsd:sequence>
                        <xsd:element name="call" type="trace:call_type"
                                minOccurs="0" maxOccurs="unbounded"/>
                        <xsd:element name="time" type="xsd:long"/>
                </xsd:sequence>
                <xsd:attribute name="name" type="xsd:string"
                use="required"/>
        </xsd:complexType>
</xsd:schema>
```

Listing A.1: Trace XML Schema Definition

A.2 Example Maven POM File

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>de.fhhof.iisys.index</groupId>
  <artifactId>h2rtree</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>h2rtree</name>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.3.1</version>
        <configuration>
          <archive>
            <manifestEntries>
              <Startup-Class>
                de.fhhof.iisys.index.h2.rtree.RTreeIndex
              </Startup-Class>
            </manifestEntries>
          </archive>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.3.2</version>
        <configuration>
          <compilerVersion>1.7</compilerVersion>
          <target>1.7</target>
          <source>1.7</source>
         </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
  </dependencies>
</project>
```

Listing A.2: Example of a Maven POM File for Building the R-Tree

Versicherung an Eides statt

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.
Weitere Personen waren an der Abfassung der vorliegenden Arbeit nicht beteiligt. Die Hilfe eines Promotionsberaters habe ich nicht in Anspruch genommen. Weitere Personen haben von mir keine geldwerten Leistungen für Arbeit erhalten, die nicht als solche kenntlich gemacht worden sind.
Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt.

Bamberg, den 16.12.2016 Carsten Kropf