Otto-von-Guericke-University Magdeburg

Faculty of Computer Science

Department of Technical and Business Information Systems

Diplomarbeit

(Master thesis)

Development of load-balanced KDB-tree

Author:

Nguyen Cong Binh

June 16, 2012

Supervisor:

Dr. -Ing. Eike Schallehn

Otto von Guericke University Magdeburg
Faculty of Computer Science
P.O. Box 4120, 39016 Magdeburg, Germany

Nguyen Cong Binh:

Development of load-balanced KDB-tree

Diplomarbeit (Masterthesis), Otto von Guericke University Magdeburg, 2012.

Contents

List of Figures

List of Abbreviations

DBA	Database Administrator
DBMS	Database Management System
DBS	Database system
CPU	Central processing unit
SQL	Structured Query Language
RAM	Random-Access-Memory

Chapter 1

Introduction

1.1 Motivation

Nowadays, database tuning plays an important role in a database system. Data-intensive applications that use relational database often face problems with the efficiency of query optimization. It affects not only the performance but also indirectly usability and scalability of many e-commerce transactions and science research (e.g., weather forecast, search engine, data mining, etc...)

Tuning a physical database design is the common and favorite choice when the database administrator accomplishes database tuning activities. Many commercial tools have been introduced to support the DBA to simplify the process of index selection, because indices can be dropped, added, and changed without affecting the database schema or application design. However it is still a difficult task to choose the right index set where the workloads change unpredictable. One of the problems with a traditional index is that the storage for storing indices increases rapidly with the volume of data. The data are treated equally whether the data is queried often or not.

A well-known multidimensional index structure for point data called KDB-tree also suffers from this problem. According to John T. Robinson [42], the storage utilization of KDB-tree structure is approximately 50-70%. That would be a storage wasting if the index is used to manage millions of data items. From the tuning concept where the memory budget is restricted, the traditional structure of the KDB-tree should be revised by adapting the access balanced tuning mechanism introduced in [15]. The goal of the concept is to ensure that the index accelerates the access to the frequently used data while keeping storage consumption a minimum. This present work is dedicated to the development of the KDB-tree that has a self-adaptation capability to the particular query situation and given storage limitation.

1.2 Goals

In this thesis, we investigate how the self-tuning concept introduced in [15] can be applied on a well-known multidimensional index: KDB-tree developed by John T. Robinson in [42].The analysis concentrate on two questions:

- How can the KDB-tree adapt its structure towards the access frequency of data and not based on the distribution of data itself?
- If the KDB-tree has the possibility of self-tuning its structure, how well does it reacts to the workload variation?

The contribution of this thesis is to answer both questions. For that purpose, first we study the original structure of KDB-tree, construct a prototype which has the same character and function as John T. Robinson proposes in the paper for a traditional multidimensional KDB-tree. Second the prototype will be modified towards the self-tuning concept proposed in [15]. Finally we conduct an evaluation to verify the implemented prototype. By using different query pattern for testing, the evaluation allows us to clarify whether the changes applied in the prototype have a positive effect on the performance.

1.3 Structure

This thesis is structured as follows. In chapter 2, the background of database as well as the query optimization process will be introduced and related to index structure. In chapter 3, the foundation of the database tuning and its basic principal would be explained. Furthermore the trend and state of the art on self-tuning index selection are also presented. Chapter 4 begins with the recapitulation of developed load-balanced index structure of preceding works and then applies the new self-tuning concept on the design of a load-balanced KDB-tree. Chapter 5 follows with the implementation of the solution approach described in chapter 4. In chapter 6 the results of the developed prototype is evaluated with different workload variations. Chapter 7 summarizes this thesis and concludes the paper.

Chapter 2

Background

This chapter contains the background knowledge to help to understand the following chapters. In this chapter the basic concepts, such as, *Database, Query Processing, Index structure*… and so on, will be introduced. Here the KDB-tree and its related function will be also briefly described.

2.1 Foundations of database systems

Based on Thomas M. Connolly in [8]: a database can be defined as a collection of related data from which users can efficiently retrieve the desired information. A database can be anything from a simple collection of roll numbers, names, addresses and phone numbers to a complex collection of sound, images, and even video or film clippings.

According to [6]: a database is a shared collection of logically related data, and a description of this data, designed to meet the information needs of an organization

Database Management System (DBMS) is an integrated set of programs used to create maintain and control access to the database. It performs certain operations like adding, updating and deleting data. The main objective of a DBMS is to provide a convenient and effective method of defining, storing, retrieving and manipulating the data contained in the database. The database and DBMS software are collectively known as database system [4]

A database system (DBS) environment consists of five components [6].

Hardware

The DBMS and the applications require hardware to run. The hardware can be a single personal computer or a network of computers. Depending on the organizations' requirements the DBS can use a wide variety of hardware with different operating systems. An important part that every DBMS needs is main memory and disk space. Choosing the right configuration of these parts could speed up the performance of DBMS

Software

The software component comprises the DBMS software itself and the application programs, together with the operating system, including network software if the DBMS is being used over a

network. These software components are written in different programming languages like Java, C++…, and use query language like SQL to access data from database

Data

From the point of view of the users the data is the most important component of the DBS environment. The database contains the operational data and the metadata (the 'data about data') and the structure of the database which is called the **schema**.

Procedures

Procedures refer to the instructions and rules that govern the design and use of the database. The users of the system and the staff that manage the database require documented procedures on how to use or run the system.

People

The final component is the people involved with the system, for example:

- Data and Database Administrators
- Database Designers
- Application Developers
- End-User

2.2 Query Processing

In order to respond to requests for information from users in a predictable and reliable fashion the database system needs a process called Query processing

Query processing: Is the activities involved in parsing, validating, optimizing, and executing a query [43].

The aims of query processing are to transform a query written in a high-level language, typically SQL, into a correct and efficient execution strategy expressed in a low-level language (implementing the relational algebra), and to execute the strategy to retrieve the required data.

An important aspect of query processing is query optimization. It deals with techniques on how to get results back in a timely manner. The aim of query optimization is to choose the most efficient execution plan between many equivalent transformation methods that minimize the resource usage for processing a query.

Query optimizers use execution time as an important factor to evaluate the performance for processing queries. The total execution time of the query is the sum of the execution times of all individual operations that make up the query [6]. There are many ways in which a complex query can be performed and therefore much research has been devoted to developing highly efficient algorithms. We can for example reduce the execution time by maximizing the number of parallel operations [28]. The optimizers must determine which execution plan is the most time and cost effective.

Query optimization uses two main techniques which are usually combined together in practice. The first technique uses **heuristic rules** that order the operations in a query into equivalent but more efficient representations. The other technique compares different algorithms applied to operations or upon the semantic within the query and chooses the one that minimizes resource usage. For example, disk access, CPU cycles, and process communications. Since disk access is slower than memory-based transfer, disk access tends to be the dominant cost in query processing for a centralized DBMS

Both methods of query optimization above depend on database statistics in order to evaluate different execution plans. The accuracy and currency of these statistics have a significant influence on the decisions of a query optimizer. The statistics cover information about relations, attributes, and indices. For example, the system catalog may store statistics giving the cardinality of relations, the number of distinct values for each attribute, and the number of levels in a multilevel index

The statistics are only helpful if they are kept up to date but keeping the statistics up-to-date can be problematic. If the DBMS updates the statistics every time a tuple is inserted, updated, or deleted, this would have a significant impact on performance. An alternative approach is to update the statistics on a periodic basis, for example nightly, or whenever the system is idle. Another approach taken by some systems is to make it the users' responsibility to indicate when the statistics are to be updated

Optimizers Are Not Perfect

The Query Optimizer is a cost-based optimizer and the quality of the execution plans that it generates is directly related to the accuracy of its cost estimations. Taking the right or wrong execution plan could lead to the difference of response time between milliseconds, minutes, or even hours.

The estimated cost of a plan is based on the mathematical model based algorithms or operators used in a query and their cardinality estimations. In order to estimate cardinality, a query optimizer needs statistical information. With the statistical information stored in the database, the optimizer reduces the amount of data that has to be processed during optimization. If the optimizer had to scan the actual table or index data for cardinality estimations, execution plan would be costly and lengthy.

Statistics are an important part in database. It can store information about distribution of records in table like histogram described above or important properties relevant to queries. It reduces or summarizes the original amount of information which may involves a certain amount of uncertainty. If a distribution statistic no longer reflects the source data due to neglected maintenance, the optimizer may make the wrong estimation about cardinalities, which in turn may lead to poor execution plans.

Even when the statistics information exists and is up-to-date, it cannot always be used properly. There is a possibility that the optimizer cannot use statistics because of poor SQL code generated by users. An expression in predicate of the query could make a bad estimation about the cardinality and obviate the optimizers to choose the best alternative.

2.3 Access structure

Before we go deeper in the next chapter to investigate the KDB-Tree and its implementation, some basic concepts and, definitions of access structures will be shown and explained.

From the point of database development, one of the main stages is database design. This stage starts only after a complete analysis of the enterprise's requirements has been undertaken and contains 3 steps:

Conceptual database design: The process of constructing a model of the data used in an enterprise, independent of all physical considerations.

Logical database design: The process of constructing a model of the data used in an enterprise based on a specific data model, but independent of a particular DBMS and other physical considerations.

Physical database design: The process of producing a description of the implementation of the database on secondary storage; it describes the base relations, file organizations, and indices used to achieve efficient access to the data, and any associated integrity constraints and security measures.

This step allows the database system running under a specific product with concrete operating system or hardware. From the user's viewpoint, the internal storage representation for relations should be transparent – the user should be able to access relations and tuples without having to specify where or how the tuples are stored. This requires that the DBMS provides physical data independence, so that users are unaffected by changes to the physical structure of the database It decides how the constraints and requirements from previous step are implemented. The other main task of physical design is choosing the file organizations and indices for the base relations so that the optimizer can select the right path to the data. Here the physical design must also face the problem of managing indices.

The index drawbacks are related to the cost of `INSERT, UPDATE, MERGE` and `DELETE` statements. When a query modifies the data in a table which was indexed by DBA, the table rows may need to be repositioned or updated resulting in the update of indices also.

So it is crucial to take into account the overhead of `INSERT, UPDATE, MERGE` and `DELETE` statements before designing the right indexing strategy. Static systems, where databases are used heavily for reporting, can afford more indices to support the read only queries. A database with a high number of transactions to modify data will need fewer indices to allow for higher throughput. Here we have to consider the disk space that indices need in order to get the better performance of the system.

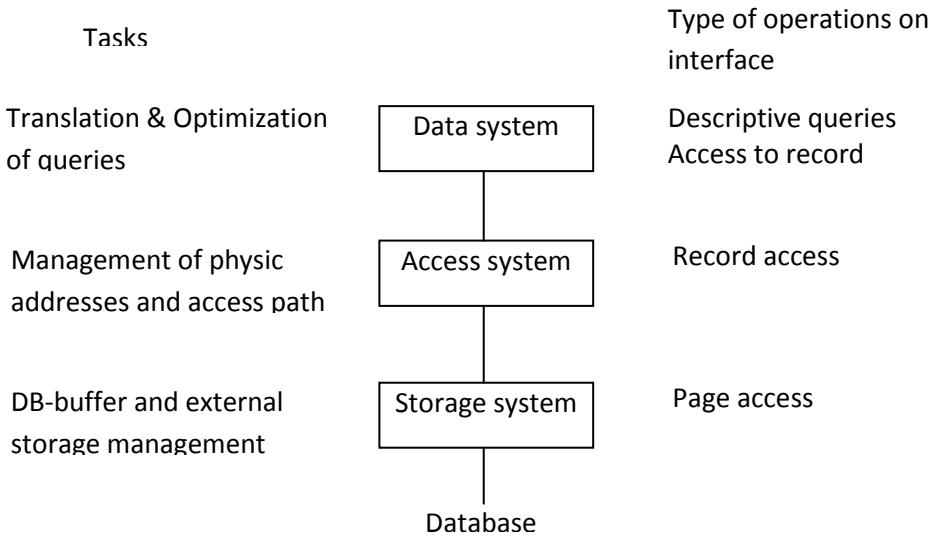

Figure 2.1: Layer model of DBMS based on [46]

The role of indices in DBMS is described more precisely by in [46]. Härder describes the structure of a DBMS in a five-layer model. Each layer represents a part of the transformation of queries from the user level to the physical storage level. Figure 2.1 shows a simplified model in which the external buffer and memory management have been integrated in the memory system.

As we can see from the figure 2.1, the access system where indices are located and managed is one of the important layers of DBMSs. Layers' role is as described in [13]:

- navigable access to the internal representation of the relations
- manipulated objects: typed data sets and internal relations, and logical access paths (Indices)
- Translation and Optimization of SQL queries

2.3.1 Types of indices

Primary index and secondary index

A file or table in database may have several indices on different search keys

Primary index: the search key of the index file specifies the sequential order of the file. It normally use the primary key of the relation but it is not necessarily so. In the primary index an index record appears for every search-key value in the file.

Secondary index: Indices whose search key specifies an order different from the sequential order of the file are called the secondary indices.

Clustered index and non –clustered index

Like primary index, a cluster index contains records that are in the same sorted form like the records of the internal relation.

When a clustered index on a column (or a number of columns) is created, the SQL server sorts the table's rows by that column(s). Since it alters the physical storage of the table, only one clustered index can be created per table.

A non-clustered index, on the other hand, does not alter the way the rows are stored in the table. It creates a completely different object within the table that contains the column(s) selected for indexing and a pointer back to the table's rows containing the data.

It is like an index in the last pages of a book, where keywords are sorted and contain the page number to the material of the book for faster reference.

The data is present in random order, but the logical ordering is specified by the index. The data rows may be randomly spread throughout the table. The non-clustered index tree contains the index keys in sorted order, with the leaf level of the index containing the pointer to the page and the row number in the data page. In non-clustered index:

- The physical order of the rows is not the same as the index order.

- Typically created on column used in `JOIN, WHERE`, and `ORDER BY` clauses.

The non-clustered index is good for tables whose values may be modified frequently.

Dense index and sparse index

A dense index in databases is a file with pairs of keys and pointers for every record in the data file. Every key in this file is associated with a particular pointer to *a record* in the sorted data file.

In comparison to a dense index that an index record appears for every search-key value in the file, the sparse index in databases is a file with pairs of keys and pointers for every block in the data file. An index record in the sparse index appears for only some of the values in the file.

All types of indices listed are referred to as the ordered indices where indices are based on a sorted ordering of the values.

Hash index

There is another type of the index called hash indices which are based on the values being distributed uniformly across a range of buckets. The bucket to which a value is assigned is determined by a function, called a `hash function`.

A hash index organizes the search keys with their associated pointers into a hash file structure. We apply a hash function on a search key to identify a bucket, and store the key and its associated pointers in the bucket (or in overflow buckets). Hash indices are most suitable for primary index structures, since if a file itself is organized using hashing; there is no need for a separate hash index structure on it.

Another classification of indexing is:

- An attribute and multiple-attribute index
- One dimensional and multi dimensional index

Index implementation: Indices can be implemented using a variety of data structures. Popular indices include B-trees, B+ trees.

Multidimensional index structures

In many applications like GIS, OLAP, the data are not usually read in a linear order. Instead it requires that data can be viewed as existing in a space of two or more dimensions. The multidimensional index should support the following queries:

- `partial-match queries`: specify values for a subset of the dimensions
- `range queries`: give the range for each dimension.

- `nearest-neighbor queries`: ask for the closest point to the given point.

One of the earliest multidimensional index structures is the KD-tree which is useful for the storage and manipulation of point objects in k-dimensional space. A split of a node is occurred by splitting a node into two child nodes according to one dimension. The dimension for that the split is chosen either randomly or according to some rules. However the KD-tree has been considered as unsuitable for non-zero size objects. An extension of KD-tree is spatial KD-tree (skd-tree) is developed to support two types of search, namely intersection and containment search.

The R-tree proposed by Antonin Guttman in 1984 is also a well-known index structure that uses multidimensional rectangles to represent multidimensional intervals. The *R*-tree consists of two different types of nodes: ***Leaf*** nodes and ***non-leaf*** nodes. ***Leaf*** nodes contain the entries of the form (***tid, rectangle***) with ***tid***s referring to records in the database. ***Non-leaf*** nodes contain entries of the form (***cp, rectangle***) with ***cp*** being a pointer to a child node of the *R*-tree and ***rectangle*** being the minimum bounding rectangle of all rectangles which are entries of that child node. The R-tree is also a balanced search tree and supports the intersection, containment, nearest neighbor search.

Another indexing approach that differs from the tree-based indexing is bitmap indexing. The basic idea behind this technique is the using of Bits. For each existing attribute value is a bit-list is created that like tuples in the fact table (data warehouse). Bitmap indexing is used specifically for data with low cardinality and with a large number of records. It supports in particular the processing of complex queries using logical operators such as AND, OR, NOT, and COUNT. The disadvantages of bitmap indexing techniques are the facts that they can be very space consuming and the insert or update operations are more expensive than for the tree-based structures.

Some multidimensional indices in the history of developing database are: Point Quadtrees, Grid File, R-tree family (R^+-tree , R^*-tree) or Arrays that can be useful for many different applications purpose. But we will discuss on the next sections the data structure that is relevant for this thesis: the partial index and the KDB-tree.

2.3.2 The partial index

In addition to the traditional index structure, a concept of partial index was presented by M. Stonebraker in [35]. Here the entire data of a table for example will not be indexed but only a

part of it. In order to create a partial index, an additional condition must be specified in SQL syntax. The SQL command could be written with the extension of WHERE- constraint:

```
CREATE INDEX <index_name>
ON <table_name>
(column_name)
WHERE <column_name> <operator> <constant>
```

When the column_name of SQL above was mentioned in a query from the user, the optimizer of the DBMS must decides whether the partial index can be used to answer the query.

For example, the manager of a company is the interested in buy volume of customers that are higher than 100,000 euro. He can create index on these customers like this:

```
CREATE INDEX BuyVolume ON Customers (Revenue)
WHERE Revenue >= 100.000
```

The partial index is created for all records in Customers-table where the attribute "revenue" is higher than 100.000 (uncommon value). A query can have benefit from BuyVolume- index, if it requests the customer whose revenue is in this interval like this

```
SELECT * FROM Customers
WHERE revenue > = 120.000
```

A query like

```
SELECT * FROM Customers
WHERE revenue > 95.000 and revenue < 110.000
```

will not take the advantage of the partial index because only a part of query is in the interval defined in the BuyVolume- index.

As seen from the example the partial indices are useful for various applications. It can be set up to exclude common or uninteresting values. Especially helpful is a partial index for creating a full index. If a table has millions of records, it is not easy to create the single index. Here, the partial index can be created stepwise on data (avoiding negative effect of “lock table”) until it covers the entire data set and thus the full index is complete.

With partial indices, the query-driven index tuning can be beneficial. By analyzing the access frequency on the column of table, the database can generate a partial index on the relevant tuples. It needs less storage and will speed up those queries that do use the index.

The disadvantage of partial indices is the lack of support for database operations such as join operation. It is profitable in case that the indexed data does not change often and the partial

indices can be recreated occasionally to adjust for new data distribution, but this entails added maintenance effort.

2.4 KDB-tree

The KDB-tree was first introduced in 1981 by John T. Robinson. The author has developed it with purpose that it will solve the problem of retrieving multi-key records via range queries. Its structure in comparison with another index structure is that it can be located in secondary memory and support queries where insertions and deletions are intermixed with queries.

The KDB-tree is a well-known extension of the adaptive KD-tree [6] and combines the properties of the B-Tree [7]. It means that with the KDB-Tree the multidimensional search efficiency of the balanced KD-trees and the I/O efficiency of B-trees are inherent in the KDB-tree.

With the Grid File and another multidimensional index structure the KDB-tree is one of the earliest space partitioning methods whose partitions are not overlapping. It has the advantage that it can apply to multidimensional points (dimension-independent fan-out). The disadvantages of KDB-tree are that it only support with point data (for non-point data, a transformation method is needed). And it does not guarantee utilization. The next section describes more details about the KDB-tree.

Type of page in the KDB-tree

The KDB-tree consists of two page types: the region page and the point page.

Region pages (index node, KDB-node)

- The region pages store the descriptions of subspaces where the data points are stored and pointers to the descendant pages. The elements in region pages (pointers and subspaces descriptions pairs) are referred to as entries. The subspaces in region pages are pair wise disjoint.
- A region page has a structure of a binary KD-tree and has a limited size.

Point pages (data node, bucket, or leaf node)

The point page is used to store the actual data or references to them. They are shown in the next figure as cylinders. Storage can be done in an unordered list, which has a size limit S. Due to the fact that the point page represents physically a memory page, it makes sense to choose a size that allows a look up of an entry in an acceptable time.

Define a point to be an element of $domain_0 \times domain_1 \times ... \times domain_{K-1}$. A region is a set of all points $x_0, x_1, ..., x_{k-1}$ if it satisfies:

$$\min_i \le x_i \le \max_i \qquad 0 \le i \le K-1$$

for some collection of $\min_i$, $\max_i \subset domain_i$. Points can be represented most simply by storing x_i, and regions by storing $\min_i$, and $\max_i$.

A range query can be expressed by specifying a region, the query region as a cross product of intervals $I_0 \times I_1 \times ... \times I_{K-1}$

If some of the intervals of a query region are full domains, the query is a partial range query; if some of the intervals are points and the rest are full domains, the query is a partial match query: if all of the intervals are points, the query is an exact match query.

Character of the KDB-tree

- Search has access **log[N/h]** which **h** is the height from root to leaf of KDB-tree.
- The index node of KDB-tree must be represented as a binary KD-tree to be dimension independent.
- The index node does not hold the data, just the leaf node.
- The path from the root to the leaf nodes (or data nodes) has the same height which means the tree is always perfectly balanced.

Function of KDB-tree

For the KDB-tree, the following functions are defined:

- Insert a point to KDB-Tree.
- Split a region page or a point page.
- Merge two region pages.

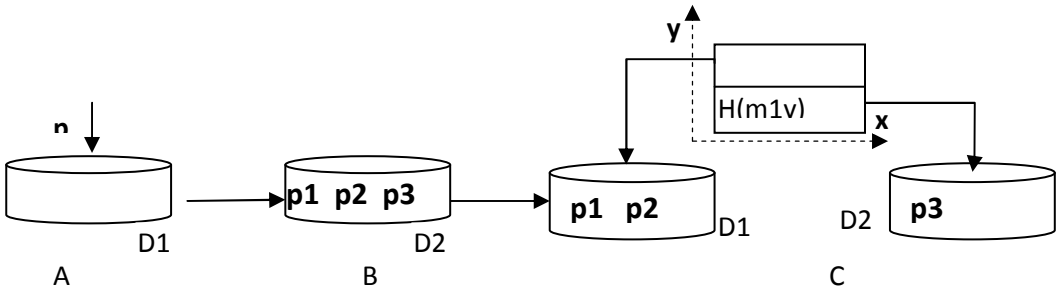

Figure 2.2: Insert point to KDB-tree.

The insert of a point into a KDB-Tree entails search and splitting strategy of another node in the tree. A search operation is performed by a top-down tree traversal. Search queries are answered analogously to the KD-Tree algorithms. Because the insert and split function on KDB-tree are described generally in books, this section below will go further in details to help the reader better understand how the balance mechanism works on KDB-tree. For explanation, a KDB-tree where region page can hold 3 subregions and a bucket can store a maximum of 2 points is presented. A point has 2 dimensions. When a bucket is full of points, it needs to be split.

The figure 2.2-A shows the state of a KDB-tree at the beginning. It contains only a data node and has no index node. The data node can only store a maximum 3 points (2.2-A). After three points are inserted and the bucket is full (figure 2.2-B); a split of this bucket is necessary. We chose a domain Y to split the bucket and the split value is the mean value of these 3 points on domain Y. Variations of how the domain and splitting value are chosen, will be discussed later .A new index node which represents a region is created to wrap these 3 points.

$$\text{Split value: } \mathbf{m1y} = (p1_y + p2_y + p3_y)\ /3$$

As described above the region page does not store the point, only data node. A hyperplane Hm1y is added to this region in order to divide this region into 2 subregions which corresponds to a set of points located on it. The points p1, p2 remains in existent bucket D1, point p3 is now shifted to a new bucket D2 (figure 2.2-C).

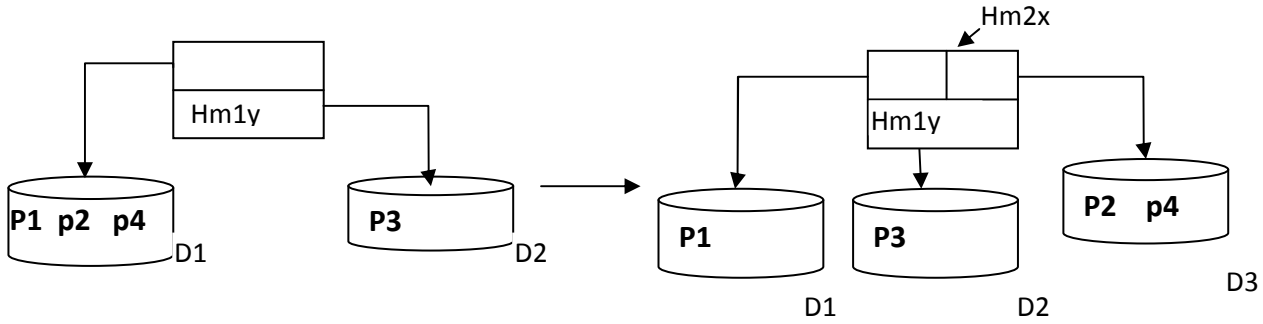

Figure 2.3: Add an hyperplane to an index node.

The bucket D1 receives new points p4 which make bucket D1 full and need to be split. A new bucket D3 is generated to store the split points. A hyperplane **Hm2x** on domain X is added to the root of tree (figure 2.4.2) where

$$\mathbf{m2x} = (p1_x + p2_x + p4_x)\ /3$$

Point p5 is now inserted into bucket D3 (figure 2.4-A). Because the index node can contain only maximum 3 subregions, a split of bucket D3 entails split of father region page. We choose hyperplane Hm3x on domain X (figure 2.4-B) where

$$\mathbf{m3x} = (p2_y + p4_y + p5_y)\ /3$$

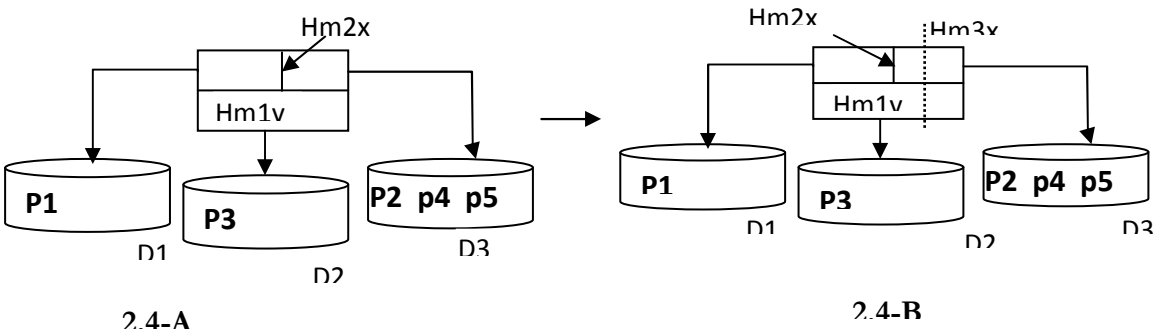

Figure 2.4: Split an index node

This split affects not only bucket D3 which is now split into two bucket D3-1 and D3-2 but also split bucket D2 into bucket D2-1 and D2-2. Points p3, p5 are shifted to new bucket D3-2 while bucket D2-2 is empty as end effect of split result:

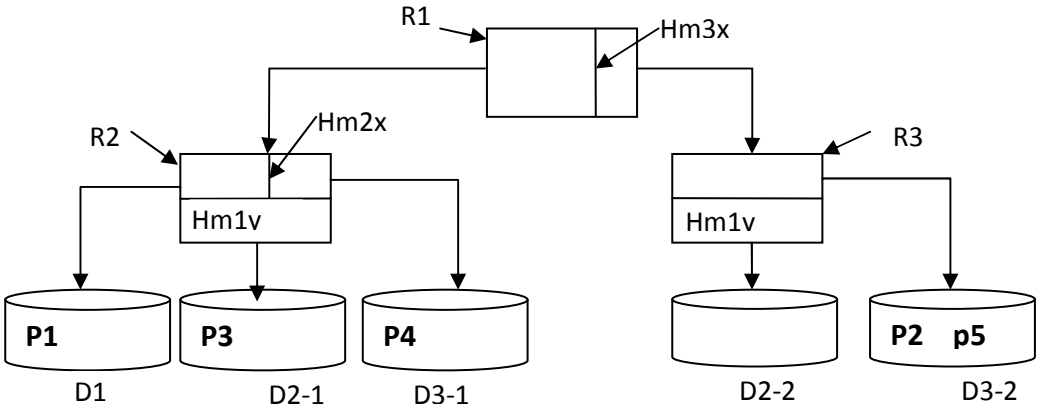

Figure 2.5: KDB-tree after splitting

.

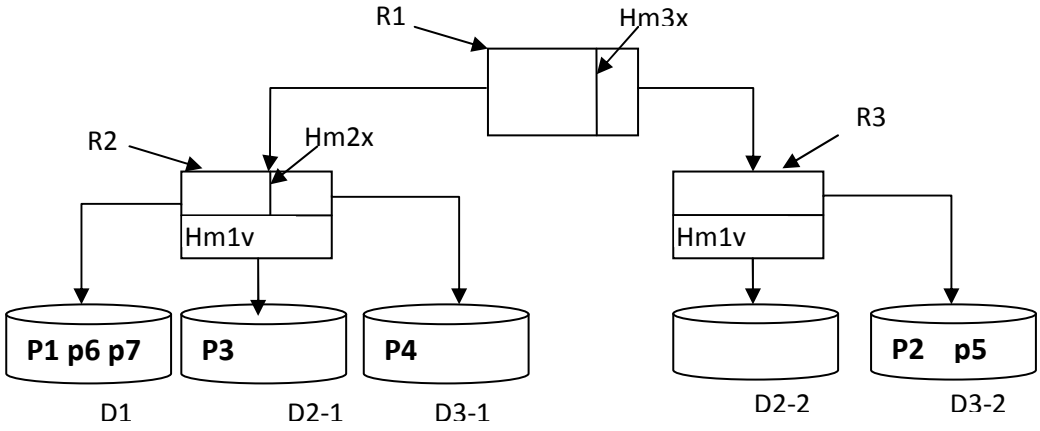

Figure 2.6: Insert point to KDB-tree

As we see each index node as seen above corresponds to an interval-shaped region. Regions which are represented by index nodes, at the same tree level (R2, R3) are mutually disjointed, their union (R1) is a complete universe (figure 2.5)

An insert of point P7 into bucket D1 make this bucket again full. The split of this bucket forces the parent index node (R2) to split because R2 does not have enough space left to accommodate new entries. Hyperplane **Hm4y** (figure 2.6) on domain **Y** is used to divide bucket D1 and region page R1 where

$$\mathbf{m4y=}\ (p1_y + p6_y + p7_y)\ /3$$

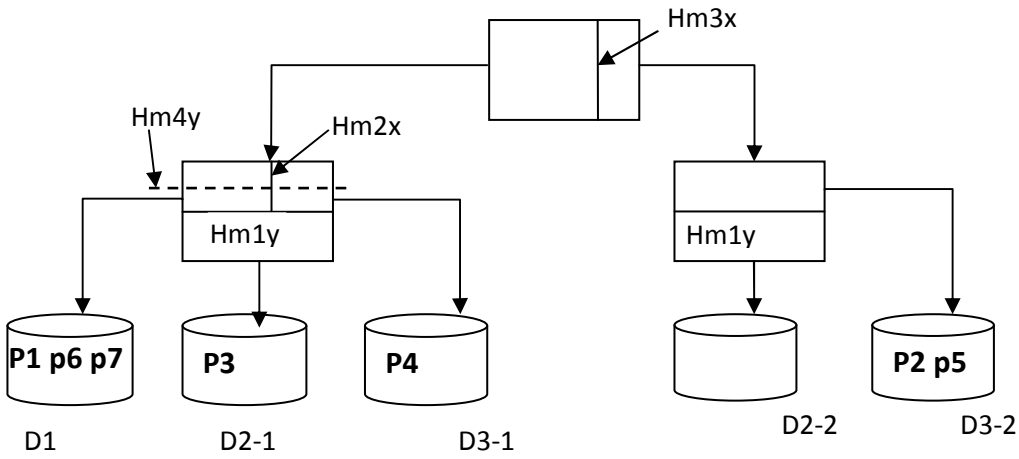

Figure 2.7: Split an index node

A hyper plane Hm4y is now added at the root of the KDB-tree (figure 2.7). The bucket D1 is split into bucket D1-1 and D1-2. The points are distributed among between 2 new subregions, depending on their position relative to the hyperplane. Bucket D1-1 stores point P1 while bucket D1-2 contains points P6, P7

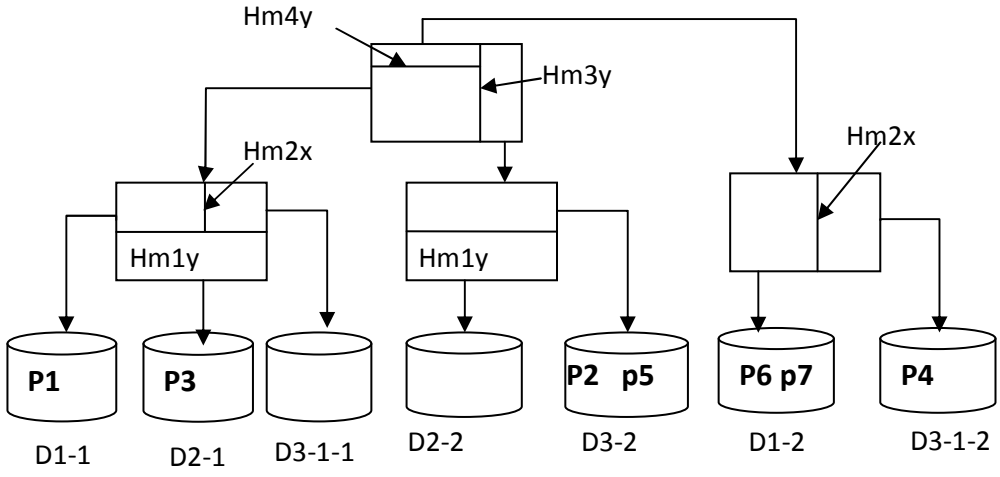

Figure 2.8: Cascading split of index node

Expecting that point p8, p9 are now inserted into bucket D1-1. Like previous step, a hyperplane Hm5x is needed to partition this bucket

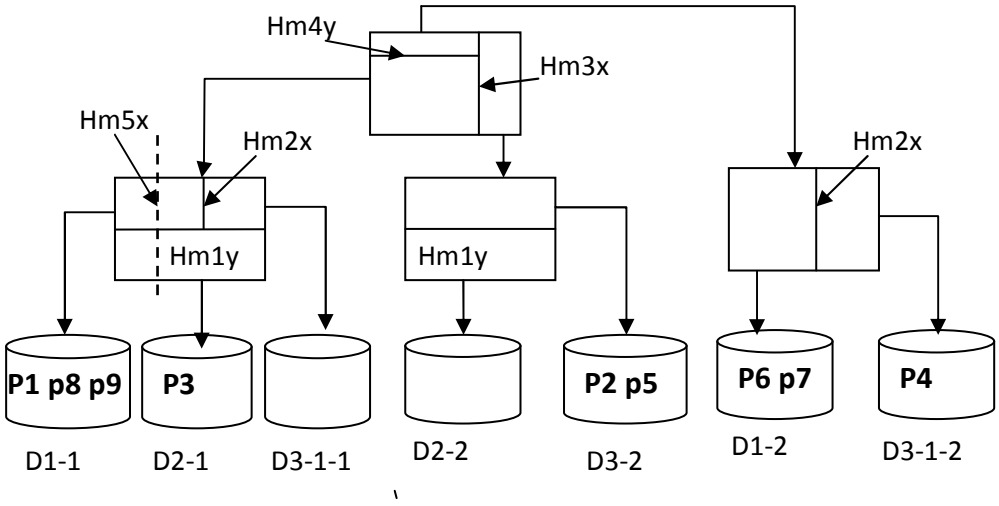

Figure 2.9: Insert point to KDB-tree

As result the KDB-tree achieve the new balance with 5 index nodes and 9 buckets. Because the root of KDB-tree has now more one entry as needed, it also needed to split. To split this root we choose an existent hyperplane: **Hm4y**

Here the region page which contains points P2, P5 intersects with the splitting hyperplane Hm4y and muss be split as well. The result of this step is a KDB-tree with 8 index nodes and 10 buckets.

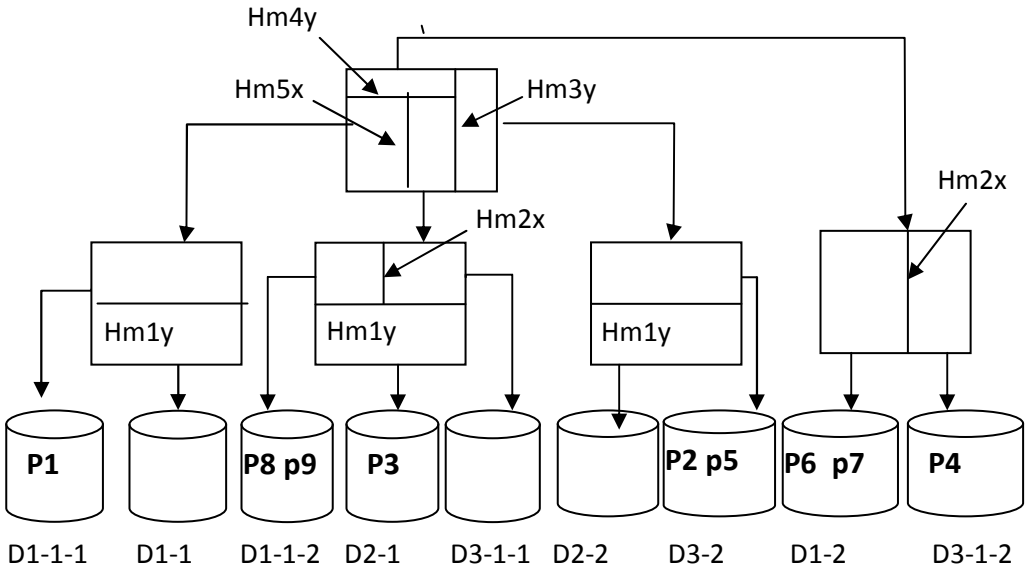

Figure 2.10: KDB-tree after splitting

Summary: The point page is split such that the two resultant point pages will contain the same number of data points. The split of point page need an extra entry of this point page in parent region page. Therefore, the split of point page may cause the parent region page to be split as well (in worst case can ripple all the way to the root) (figure 2.10).

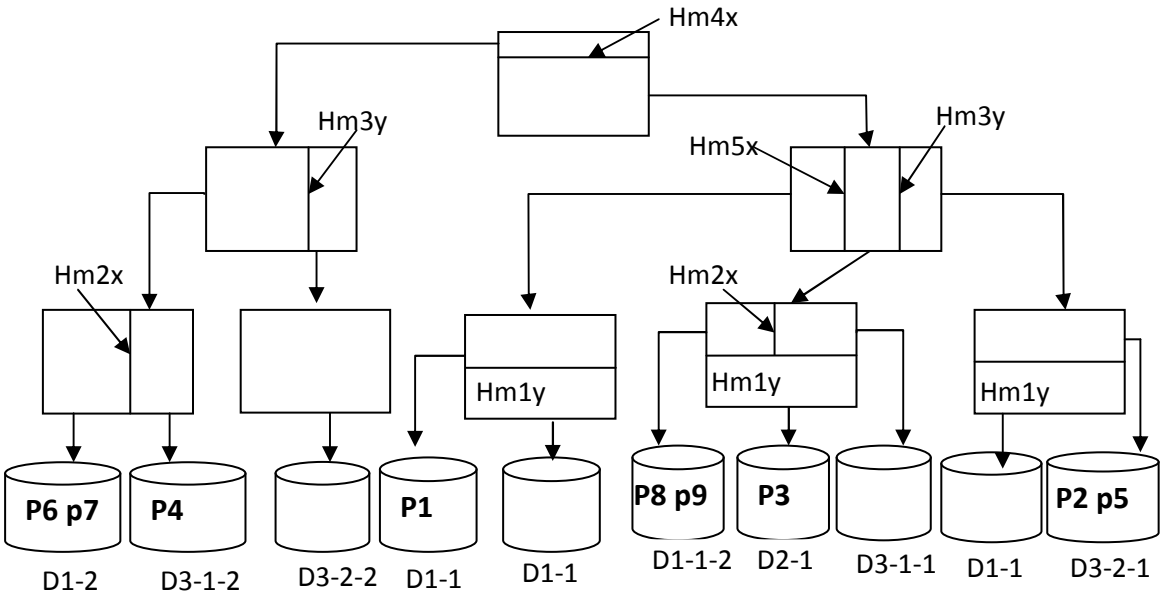

Figure 2.11: KDB-tree after splitting

The split of the region page again may also affect index node of lower level of the tree, which has to be split. When a region page is split, the entries are partitioned into two groups. A hyperplane is used to split the space of region page into two spaces and this hyperplane may cut across the subspaces of some entries. Downward propagation of split may occur as the consequence. Because of this forced split effect, it is not possible to guarantee a minimum storage utilization.

As a result, updates can be very inefficient and, maybe more importantly, the space utilization can decrease dramatically since the split process may generate many near empty leaves. In addition the reorganization effort to balance the tree is complex. The structure of the KDB-tree, as proposed, cannot be used for non-zero sized objects like lines, or regions.

Merge strategy

The merge strategy always looks up two neighbor rectangle to merge if the buckets pointed to these two rectangles are underflow.

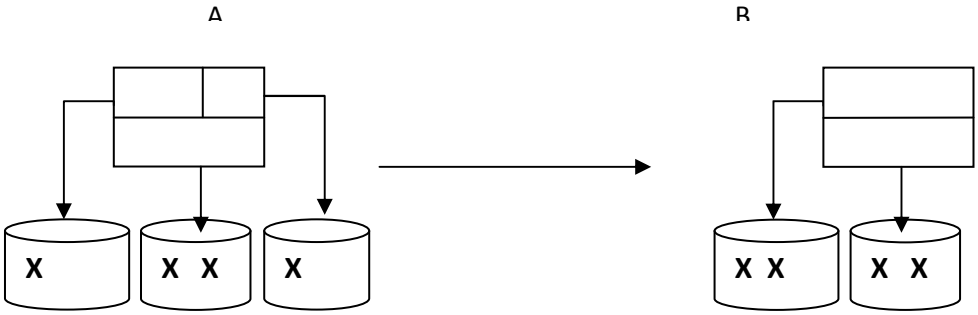

Figure 2.12: Merge two subregions

As we can see the two internal nodes (figure 2.12-A), each contains only one point that is stored on the data node that they point to. Because a data node can store 3 points, we can merge two subregions into one subregion (figure 2.12- B)

But there is a case where the condition of merging 2 neighboring internal nodes is not fulfilled (figure 2.13 A1), but rectangle which contains these two internal nodes is met. The next figure shows that case.

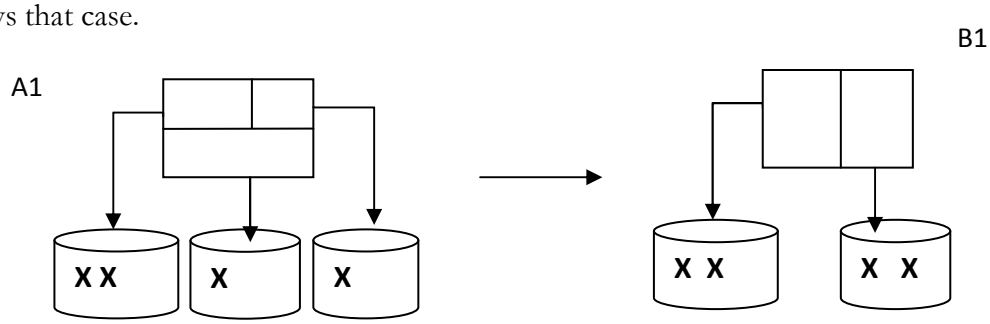

Figure 2.13: Merge two subregions

As we can see the rectangle storing three rectangles has the total number of entries that can be divided into two buckets. In this case the reorganization of the rectangle of higher level is necessary. Two or more pages whose data spaces form a rectangular space can be merged and the resultant page is resplit (figure 2.13 B1).

The procedure to find a suitable sibling node to merge with may involve several nodes. The union of data pages results in the deletion of at least one hyperplane in the parent index node. If an underflow occurs, the deletion has to be propagated to the tree.

Variations of the above algorithm result from the way $domain_i$ and x_i are chosen in (14). One way of choosing $domain_i$ is to do so cyclically, as follows. Store in each region page a variable `splitting_domain`, initialized to 0 in a root page when a new root page is created. When a page is split, an element of $domain_{splittingdomain}$ is used, and the new pages have `splitting domain` set to (`splitting_domain + 1`) `MOD K` (figure 2.14).

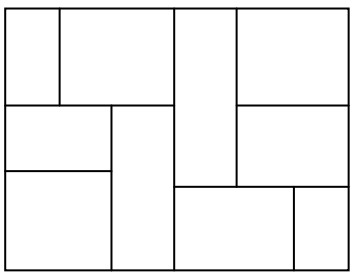

Figure 2.14: Cyclic splitting pattern

This cyclic method might be modified if something is known about queries. For example, suppose K=2, and that "most" queries are partial match queries or partial range queries on $domain_0$ only. In such a case it would be desirable to use $domain_0$ several times in a row before incrementing splitting domain, resulting in a splitting pattern like figure 2.15.

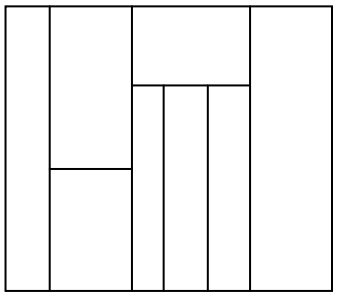

Figure 2.15: Domain X priority splitting

Chapter 3

Database Tuning and Self-tuning

This chapter contains an introduction to the database tuning and the aspects that can affect the tuning performance. In second part, we describe how the automated database tuning can be design with different approach. Subsequently some related works on the load-balanced access tuning mechanism are presented. Finally the recently research on database self-tuning will be discussed.

3.1 Foundation of Database Tuning

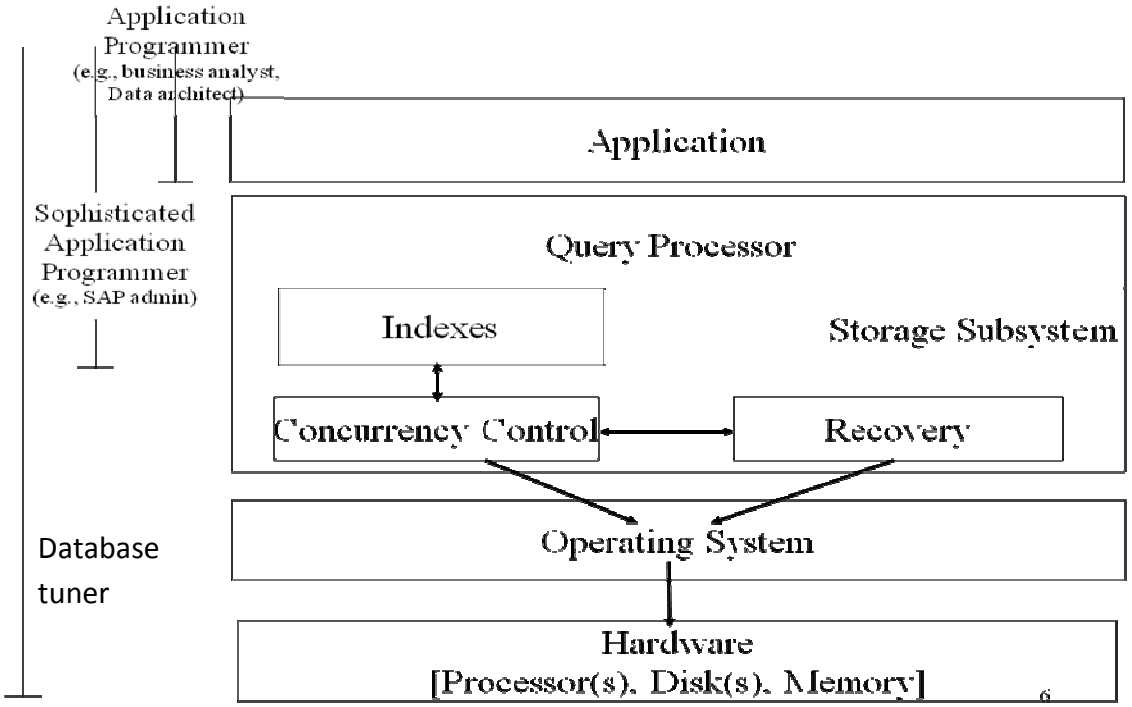

Figure 3.1: Database tuning tasks [9].

According to Dennis Shasha in [9], database tuning in defined as follows:

"Database tuning is the activity of making a database application run more quickly. "More quickly" usually means higher throughput, though it may mean lower response time for some applications. To make a system run more quickly, the database tuner may have to change the way

applications are constructed, the data structures and parameters of a database system, the configuration of the operating system, or the hardware".

Database tuning requires very broad skills and comprises the fields of system administration, database administration and application development. Figure 3.1 shows the range of possible database tuning tasks. It refers to the design of the database files, query tuning, selection of the database management system (DBMS), operating system and CPU the DBMS runs on. Problems can also arise at the application level.

Performance can be understood on several aspects such as: quality of processing (and results), availability, usability, etc. This is already a challenge in itself since diverse factors can affect the overall database performance.

Database tuning mostly refers to runtime performance and related aspects, e.g. [33]

- **Throughput:** number of queries or transactions that can be processed in a fixed time
- **Response time:** time from initiation of single query until full result is available
- **Resource consumption:** temporary (e.g. CPU, memory) or constant (e.g. hard disk) use of resources

Depends on the operating system, existing hardware and another factors, tuning the database performance is not a simple task. Usually when the users complain about performance problems, it should be a hint to tune the database. In practice there is no "one fits it all" tuning approach. Hardware upgrade for example would be an easy choice when the system runs slower than expected but causes more cost than index tuning approach. If an application cannot get all the resources that it demands, the application may run slowly. If many applications demand a particular resource and the database system cannot fulfill all demands, the resource becomes a bottleneck. In general the tuning of the database is to avoid obvious slowdowns and balance the available finite resources (I/O bandwidth, memory, disk and CPU).

Like optimization activities in other areas of computer science and engineering, database tuning must work within the constraints of its underlying technology. It would be ideal if no resource becomes a bottleneck when applications are being executed. However due to other constraints, like internal operating system limits or policies such as resource was used by another application, there is no guarantee that bottlenecks will not happen. While the database tuners cannot change the underlying database management system, they can modify table design, select new indices, rearrange transactions, tamper with the operating system, or buy hardware. To detect the performance problem, the DBA needs to collect regular performance data. This information is used as an historical benchmark to find where the problem may occur. The goals are to eliminate

bottlenecks, decrease the number of accesses to disks, and guarantee response time, at least in a statistical sense. This can range from increasing the number of resources in a system (by buying additional hardware) to altering the system's configuration so that existing resources are utilized in an optimal manner. Sasha and Bonnet have for tuning DBMS placed in [9] five tuning basic principles:

- Think globally, fix locally (Localizing the problems)
- Partitioning breaks bottlenecks (temporal and spatial)
- Start-up costs are high; running costs are low
- Render onto server what is due onto Server
- Be prepared for trade-offs

Before tuning methods are applied, the performance goals need to be reasonable. There should be sufficient information to anticipate or track performance statistic before taking action. Another approach could be using alert monitoring tools. For example the DBA can set a threshold value such as maximum response time for the transaction. The application would automatically notify the DBA if a query response time exceeds the threshold in the production environment. Once a performance issue was detected, the next step is to identify the bottleneck in system. For example if the performance affects only one application, the reason could be SQL code, or all application with a given table have similar problems, the root of bottleneck can be locking or indexing on that table.

3.2 Automated Database Tuning

The database management system (DBMS) as described in chapter 1 is a suite of services (software applications) for managing databases, which enables simple access to data and manipulates the data found in the database. The Database Management Systems (DBMS) therefore must function efficiently and should have tools that support the system in case of partial failure or system resource bottlenecks. It must ensure to maximize the use of system resources to perform work as efficiently and rapidly as possible. The performance of a database system is impacted by factors for example data size, workload, and amounts of users or user processes. All these factors degrade the system response time. Traditionally, the database administrator (DBA) was responsible for tuning the system to ensure optimal performance. In reality it is not an easy task for the DBA to follow the system's performance and identify the causes. The only practical solution is to make a database system self-tunable so that tuning proceeds automatically with minimal human intervention [39].

A database system is considered to be autonomic if it is self-configuring, self-optimizing, self-healing or self-protecting. These aspects were defined by IBM for an autonomic self-management system [23]:

- **Self-configuration**: Automatic configuration of components;
- **Self-healing**: Automatic discovery, and correction of faults;
- **Self-optimization**: Automatic monitoring and control of resources to ensure the optimal functioning with respect to the defined requirements;
- **Self-protection**: Proactive identification and protection from arbitrary attacks.

Instead of controlling the system directly, now the intervention of the human operator on the system is to design general policies, rules, constraints that administer the system so there is automatic adaptation to external influences, with the aim of improving performance of the system and achieve the goal that the database is easier to use and maintain.

In [40] the important reasons for a self-tuning system are listed. High personnel costs for specialists, large number of parameters in database and the multiple layers of DBMS could lead to lose control overall system.

Self-tuning is a concept borrowed from Control Theory. Feedback loop design and analysis is one of the main tasks in control theory. Its theory provides a systematic approach to designing closed loop systems that are stable in that they avoid wild oscillations, are accurate in that they achieve the desired outputs (e.g., response time objectives), and settle quickly to steady state values (e.g., to adjust to workload dynamics). The feedback control loop (FCL) has a wide implementation in computing systems and networks. In comparison to feedback control loop in Control Theory, the authors in project COMFORT [12] recommended an approach which divided the feedback control loop into three phases: observation, prediction, and reaction.

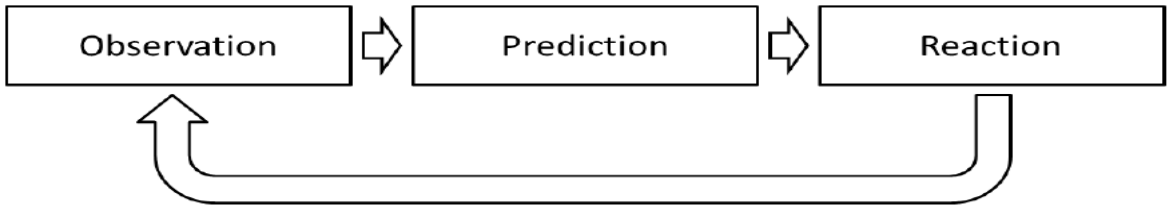

Figure 3.2: Feedback control loop for self-tuning

In [2] these steps of self-tuning are referred to as: monitor, assess and reallocate. A DBMS must constantly monitor itself to detect if performance metrics exceed specified thresholds. When a problem is detected, the system must assess various resource adjustments that can be made to solve the problem. Finally the DBMS must relocate its resource to solve the problem.

An alternative approach that has the similarities with the FCL is MAPE developed by IBM. MAPE stands for monitor, analyze, plan and execute. These four phases are operated sequentially. The first phase Monitor corresponds to the Observation phase in FCL. The data obtained from this phase are analyzed in Analyze phase and it decides whether a change of parameters is necessary. In phase 3, an action plan based on information from phase 2 is constructed. This plan will be executed in phase 4. All phases of MAPE have access to the knowledge database containing information and solutions. According to author in [44] not only the parameters are considered as adjustable indicator for software, but each discrete element in a workflow, hardware can be also monitored and regulated by MAPE.

3.3 Index self-tuning

The performance of a database system depends crucially on its physical design, i.e. the set of physical structures such as indices and materialized views. As physical data design is an independent element in relational DBMS architecture, change in physical structures such as index does not affect the result of the query, in contrast it can speed up the execution of the query. In comparison with another tuning method, index tuning is still a favorite approach that requires less effort and cost.

Traditional physical design tuning assumes that there is a representative workload in normal services of database. The query load is rather stable and the DBA tuner muss analyze and find out certain query characteristics that are predominant in the workload. Under the assumption that this representative workload would be submitted to the database in the future, the DBA tuner can maximize the overall performance by manually choosing indices or using an automated tool like Index Tuning Wizard in Microsoft SQL Server or DB2Advisor for DB2. The majority of methods for this kind of index selection are off-line approaches.

It seems this paradigm is "passive" in the way that it has limited capability in automatically detecting a pattern in query workload and in many cases must require human intervention. This is not suitable and optimal for scientific data management, where queries can result from interactive data analysis and their characteristics exhibit the infrequent trait.

While the off-line tuning approach is still widely used in many vendors' databases, the computer scientists now focus more research in another direction: on-line tuning techniques. To adapt to trends of query workloads which are unpredictable and time-dependent, some studies have developed rudimentary on-line tools for index selection in relational databases. It promises a high

potential benefit for database administration when this approach overcomes the problem caused by on-line selection that is never faced in off-line tuning.

The main idea is to add an on-line tuning module to the database architecture that monitors the query load to identify dominant access patterns, and automatically adjusts the index configuration to maximize query performance. This is a challenge that is not found in off-line setting because on-line tuning must indentify promising configurations of an index based on current trends in the queries. It must take decision when the new index configuration should be materialized. To do that an online tuner must have enough statistical information and a cost estimation algorithm. These aspects need more analysis and research. Furthermore, on-line tuning must execute in a productive environment with limited overhead and therefore has to avoid disrupting the concurrent execution of queries.

The problem of online index selection is to choose for each q_i a configuration $C_i \in \boldsymbol{S}$ that minimizing the combined cost of queries and configuration changes [18] while processing the query q_i . Here $\boldsymbol{S}$ contains all sets of indices that can be defined over the existing table and `size`($\boldsymbol{S}$) does not exceed a given size limit.

$$TotWork = \sum_{i=1}^{n} \delta(C_{i-1}, C_i) + \text{cost}(q_i, C_i)$$

Where:

- cost(q; C) for the cost of evaluating query $\boldsymbol{q}$ under configuration $\boldsymbol{C}$
- $\delta\ (C_{i-1}, C_i)$ is the cost for changing between two configurations $C_{i-1}, C_i \in \boldsymbol{S}$

Without knowing the queries performed in the future, the online algorithm must select each configuration C_i for query q_i by evaluating in the past executed queries ($q_1, q_2, ..., q_{i-1}$). Here the cost for analyzing the observed queries ($q_1, q_2, ..., q_{i-1}$) must be considered because it can cause resource overhead. It would be a challenge of online index tuning to evaluate **cost(q; C)** efficiently.

Proposals for On-line Tuning

AutoAdmin Project [32]

One of the early on-line algorithms was published by Bruno and Chaudhuri during the AutoAdmin Project at Microsoft Research [32]. In contrast to another algorithm, it takes a retrospective approach. In order to suggest a set of indices that suitable for one representative

input workload it analyses independently every candidate's single index that may be created or ejected for that workload. If there is a benefit where an index can improve the performance, that index is taken into consideration. The algorithm must also identify the point where this index change has been beneficial in hindsight. To apply this step to a set of indices, parallel running of multiple single-index algorithms is required. The authors also suggest a benefit metric to make comparisons of two indices where index interactions between them can happen.

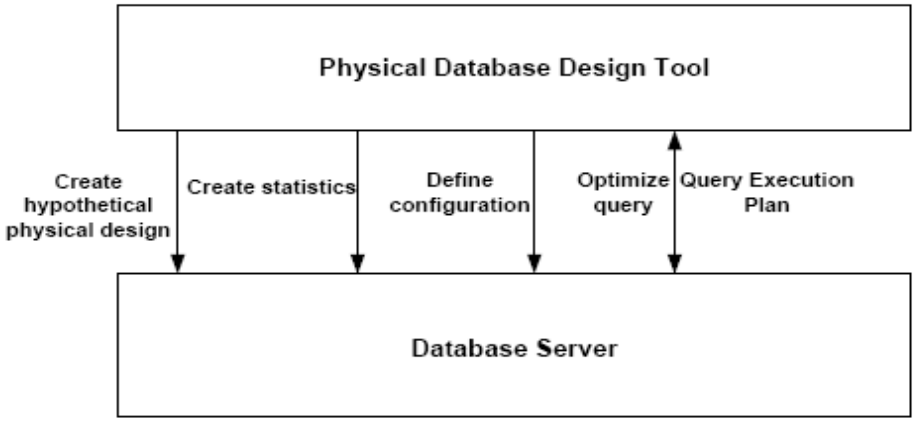

Figure 3.3: "What-if" analyze architecture for physical database design [27]

This approach follows one of principle self-tuning "what-if analysis" to obtain benefit estimates for each (query, index) pair.

COLT Project [17]

COLT which stands for Continuous On-Line Tuning is a system for on-line index selection. In this system predictions are made relatively frequently (10 queries in experiments for each period). The queries whose performance relate to indices are taken into consideration. The future benefit of index candidates is computed after each period by measuring the efficiency of each index against a sample of queries from the query space above.

To restrict the analysis index space, indices are divided into three classes: (a) the materialized indices, (b) the hot indices that are not materialized but are considered promising, and (c) the remaining indices.

For each index set the authors have a different strategy. Hot indices are indices that could be materialized and therefore are put into what-if optimization to evaluate their potential gains. For the less promising indices, on the other hand, Colt uses much simpler estimates of their performance. Moving indices between these classes based on computing their gain expectations; hence, Colt may shift indices from lower class to the materialized set (index creation), remove

some from materialized (index deletion), and then update the set of hot indices based on the accumulated statistics.

The evaluation metric for predicted benefit of index is computed using a different approach. "The basic idea behind this forecasting model is to apply a smoothing procedure to the past benefit measurements, then conservatively predict how the recent trends will continue. The most important difference in this prediction model is that the metric is meant to be directly comparable to the cost of materializing the index. This is important because the predicted benefit and materialization cost may be subtracted to predict the net effect of creating the index." [18]

QUIET [15]

One interesting approach to on-line database tuning is a system called QUIET. Similar to COLT, QUIET predicts the future benefit of individual candidate indices. The algorithm falls into the predictive class of on-line algorithms. Throughout the operation of the system, QUIET maintains a pool of candidate indices that are either materialized already or appear promising to be created in the future. The size of this pool is configured by the DBA as a system parameter and act as a persistent index cache. The algorithm periodically predicts the benefit of each candidate index, and uses the prediction to select a set of indices with the most benefit overall, subject to a limit on the total index storage. Unlike COLT, a threshold T defined by user is applied in QUIET in order to decide when an index can be materialized. These indices replace the current index configuration if their predicted benefit is larger by a factor of at least T. Quiet distinguishes between materialized and virtual (i.e. currently not materialized) indices.

One of the interesting aspects of QUIET is the function for future benefit. For each candidate index, QUIET tracks the most recent k queries that would be affected by the index. The benefit of the index for each query is scaled based on the age of the measurement, and the scaled benefits are added. For this, the index recommendations (including the expected Profits) are provided with time stamps (i.e., for example, a monotonically increasing Request counter). The total profit for a period that ends at the time t_{SE}, with the recommendations I made inquiries to an index with the time stamps $t_{S1}, t_{S2}, ..., t_{Sk}$ then calculated from (where $t_{S1}, t_{S2}, ..., t_{Sk}$ are the timestamps of queries in the history of I, the current timestamp is t_{SE}, and profit(I; t_{SJ}) is the reduction in cost that I provides for the query at timestamp.) The final prediction benefit (I) is written as

$$benefit(I) = \sum_{j=1}^{k} \frac{profit(I, t_{SJ})}{t_{SE} - t_{SJ}}$$

Another contribution of this work is the study of a technique called "on the fly" index generation. The general idea is to efficiently create the indices selected by the on-line tuner by integrating index creation with query execution [18].

Since the QUIET system is merely an extension of DB2 and not integrated, it generates a certain overhead. A further integration of the self-tuning indices in the DBMS was proposed in [24].

3.4 Related work on load-balanced index structure

3.4.1 Load - balanced index [15]

In [15] Sattler and Schallehn describe a binary tree (figure 3.4), which can be balanced by the frequency of requests on nodes. It is also possible that the number index node can increase or decrease as a result of a balanced reorganization. If the number of nodes in the binary tree exceeds the maximum allowed nodes, the entries would be relocated into a page container. This page contains the leaves of an inner index node and acts as a list. Excess entries will be stored there in case the tree cannot create more nodes. If the number of index nodes is reduced due to limited memory for example, the tree dissolves two nodes and their respective page container drop on the parent node. In contrast if the number of allowable nodes increases, the tree can be expanded by splitting an existent node into two successor nodes.

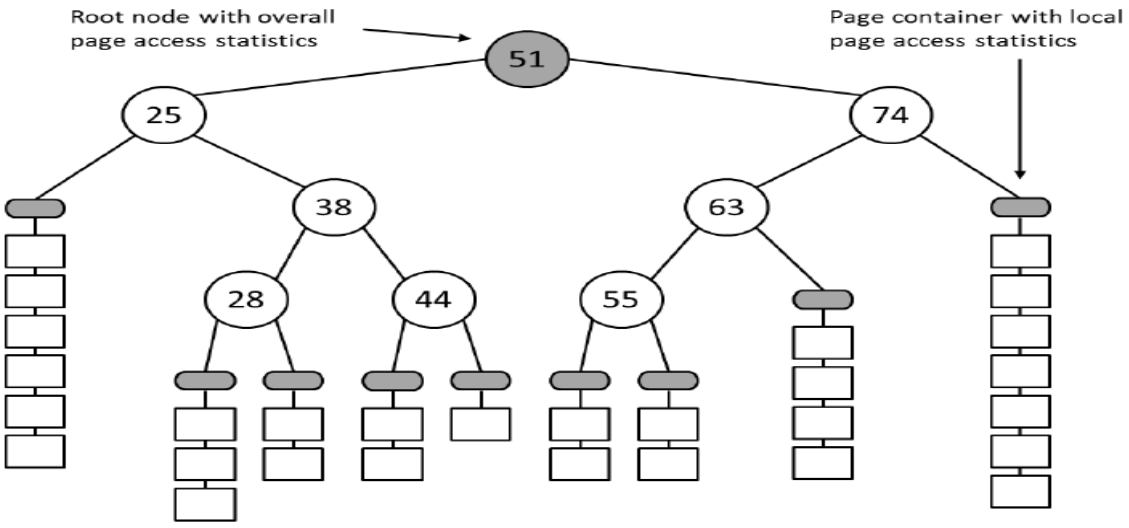

Figure 3.4: An access-balanced binary tree [15]

To demonstrate the concept, a load-balanced binary tree is implemented by the authors. This has a controllable size: namely the number of nodes. It also has a page container structure that contains data. The look up in the tree operates like an original binary tree, but within a container a sequential scan is needed. Figure 3-2 shows the tree.

The reorganization of the tree is based on two simple rules. While the root node contains the total number of requests and the actual size of the tree, and the page container stores the page accesses. The balance value is determined:

$$balance = \frac{all\ page\ accessed}{number\ of\ page\ containers}$$

For nodes that refer to two containers, the amount of accesses on both containers is larger than the balance value

$$pageAccesses_{(pc1)} + pageAccesses_{(pc2)} < balance$$

Otherwise, the node is removed and added to the pool of free nodes and the two containers along that node are merged into one. Conversely a container is split into two if the access counter is twice as high as the balance value

$$pageAccesses_{(pc1)} > 2 \times balance$$

In addition, we need a free node from the pool of free nodes which is maintained as a number of free nodes in the tree root.

The experimental evaluation with 100,000 tuples, approx.100 tuples per page and tree with 1000 nodes showed that the access-balanced tree outperforms the fully balanced tree with a linear factor. After 100,000 lookups the tree reaches a relative structure and rarely needs to be reorganized. The approach opens new possibilities of extending commercial DBMS but the integration of this tuning concept in commercial solutions will not be easy. Issues such as concurrency or updated statistics within the index structure are not clarified. Operations like range scans, sorting and ordered scans, merge joins and their implementations with new kind of index must be investigated. Binary tree based access-balancing is not the optimal choice for a DBMS index structure but the tuning idea behind that option motivates the transfer of this implementation to a multidimensional index structure: KDB-tree. The next section will describes how a KDB-tree can be adapted with this concept.

3.4.2 Load balanced B-tree developed by Sebastian Mund [25].

To balance the B-tree according to the frequency of access, the statistical information is taken from every node of the tree. The access on every node is equally distributed or balanced on every child of the node if:

$$balance = \frac{all\ queries}{number\ of\ node\ pro\ \text{group}}$$

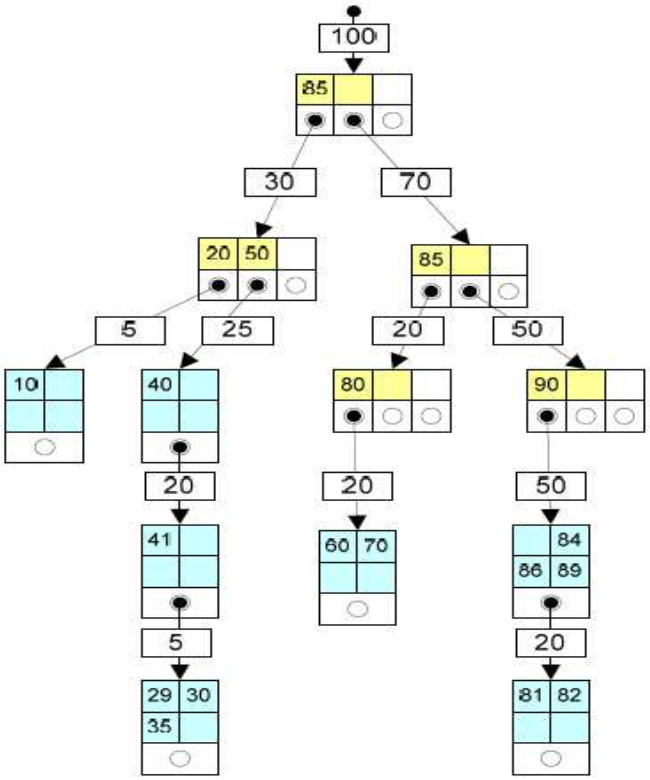

Figure 3.5: Load- balanced B-tree [25]

With the condition that the number of nodes is fixed and using this form, the author first tried to get the free node by merging or transforming the node whose balances are lower than the total access within the group (the node and its children node form a group). Afterwards the split process is executed with the node that has a higher balance access based on the free available node. The transformation and joining process always occurs before the split process because it must gain the free node. Of course with this method the b-tree is not data balanced like a traditional B-tree but more in the direction of access balance.

To get more understanding about how this tree balanced access, read the thesis of Sebastian Mund written in German language [25].

3.4.3 Load- balanced Grid-file developed by Jan Mensing [20].

Like the B-tree, to develop the load-balanced grid-file there must be the pre-condition that the size of grid-directory is fixed and cannot be increased. A new bucket is appended to another bucket when the bucket is full of entries. In this case the two-disk principle access is not guaranteed any more. In order to join two cells or split a cell in a grid-directory the author uses a

correction value in each interval of scale in the grid-directory to detect a grid-cell with the highest access and the grid-cell whose access value is the smallest. Moreover, to split an interval with highest access, the number of accesses in this interval must be higher than a breakup value, which is chosen by the author (this requirement assure that the grid-file does not need unnecessary reorganization). The author's approach showed a good result in an experiment and can be applied to real applications. It just has a weakness that the breakup value cannot be automatically chosen, and must be designated by the DBA. In general, his method is always a tradeoff between deleting a cut-off value in a scale and inserting a new cut-off value in another scale.

More information and results about his work can be found in the thesis of Jan Mensing [20].

3.5 Summary of the state of the art and related work

The path from traditional tuning to a self-tuning system has reached a relevant progress but still needs more research. Besides the valuable work with remarkable results, it still faces some challenges with new problems, and opens new research topics.

First of all as described coarsely above each algorithm chose the different approach and evaluation metric to optimize the query process. The AutoAdmin project for example chooses a representative workload to create new index while another approach like Quiet or Colt observe the queries in period in order to detect the trend of queries und predict the most promising indices. Here choosing a suitable workload is still contentious. A simple solution would be to pick only a small random sample of the workload or than divide it in period or observe the most recent queries. Another solution reduces the analyzed queries by compressing workload and leaves only those essential to solve the online ISP accurately and efficiently [29].

While AutoAdmin project chooses its own method to determine the benefit of an index, QUIET uses extra invocations of the query optimizer. Which techniques are used and how it is integrated with query optimizer for evaluation offers potential indices benefits from new research alternative in the future. Under the assumption that the storage is limited, on-line tuning still lacks the metric to evaluate index interaction since these indices are even attractive candidates in configuration and creating or deleting of one index can impact the benefit of another. It entails a question: should storage (example memory to store materialized index) is unvaried during on-line tuning process or can be extended based on workload pattern? This problem also connects to one important aspect of self-management namely self-configuration.

For column oriented DBMS, a new kind of index self-tuning has been studied and has gain attention called: database cracking. It creates and refines indices automatically and incrementally, as a side effect of query execution. It focuses on adapting the physical database layout for and by actual queries. When a column is queried by a predicate for the first time, a new cracker index for that column is initialized (in [37], [38] cracker index is an AVL-tree) and a copy of column is create (call C_{copied}). The original column is untouched in order to allow fast reconstruction of records. As the column is used in the predicates of further queries, the C_{copied} is refined and reallocated physically by range partitioning until sequentially searching a partition is faster than binary searching. The cracker index is used to localize the position value mentioned in query predicate.

Another technique contrary to database cracking is adaptive merging. When a column is request in a query predicate, the database produces sorted runs on that given column and stored in a partitioned B-tree [5]. The next queries upon that same column perform merge steps. Each merge step affects key ranges that are relevant to actual queries, avoiding any effort on all other key ranges. This merge logic executes as a side effect of query execution.

Both methods (database cracking, and adaptive merging) mentioned above for column oriented DBMS are called: adaptive indexing. Index creation becomes an integral part of query processing. The actual query execution operators and algorithms are responsible for changing the physical design "Essentially, indices are built selectively, partially and incrementally as a side-effect of query execution. Physical changes do not happen after a query is processed. Instead, they happen while processing the query and are part of query execution" [38].

Benchmarking Online Index-Tuning

To test the performance of on-line tuning, the authors in [14] recommend the four workloads suites to test how online tuning algorithms react when parameters settings vary for these workloads.

Reflex: this workload analyses the performance of online tuning with different length of workload in order to test the reaction time of algorithm. The queries used in the test involve mostly multi-column covering indices that may interact.

Queries: the second suite of workload focuses on the complexity of workload where different kinds of queries are considered from simple index to multi-column covering index, index interaction and join indices.

Updates: verifying performance of a tuning algorithm for workloads of queries and updates. For that database is tested with different stage: queries heavy or update heavy. The intuition behind that is to compare the benefit from materialized index created during heavy queries with maintenance cost of those indices while updating.

Convergence: with a stable workload that does not contain any shifts, this evaluation metric tries to diagnose the point where the online-tuning algorithm will converge to a configuration that will remain unchanged by the end of the workload.

Variability: The last suite of workload examines the performance of online tuning as the database expands the set of attributes that carry selection predicates.

In [18] a new benchmark metric is introduced for online indexing techniques. In comparison to traditional benchmarks where the cost of execution query is dominated, the online tuning must capture the cost of index creation where new index structures are intended to be created on-the-fly. Another metric could be used to evaluate "how fast the system recognizes which indices to build" [18]. The faster this happens, the more queries can benefit from an improved performance.

Chapter 4

Problem and solution approach

In this chapter, the traditional KDB-tree will be redesigned towards the load-balanced access mechanism recommended in [15]. Based on the solution approach of load-balanced access from chapter 3, we describe how the components of the KDB-tree should be modified to adapt to the new tuning concept. It also explains how the statistical information is gathered and analyzed during the tuning process to achieve the access balance goal.

4.1 Study of the problem field

As explained in the previous chapter, the use of an index structure for a traditional database is always a balance between available space in main memory, the computational capacity, and speed of the database operations. The more the primary memory can store the indices, the more the choices the query optimizer has to process the query operations. On the other side, the memory in the database is limited and usually reserved for another application. In addition the processing capabilities that are required to manage and update the index are also high. In general for a database it is more efficient and cheaper to use an index tuning method in order to access data. However, the side effects of the index tuning approach must be considered, such that while improving the data access the tuning approach does not exhaust the database resources.

Some index structures have been used widely for many years in the database access process and have played an important role in the database choice, but it still has some disadvantages. One famous access structure such as the Hash table can only manage a limited number of data (with dynamic hashing, extra level of indirection in the bucket address table is needed). As the data grows over time, this index structure has more frequent collisions, preventing an efficient search. B-trees, however, grow very fast with larger amounts of data. In addition to updating the B-tree, the index requires more space to store the larger index size. The number of B-tree index nodes increase logarithmically with the volume of data. As a result more nodes are needed to search which leads to the increasing of the search effort. Also the effort of index restructuring is also increasing.

The grid file is similar. The more data the Grid-File has to manage, the number of buckets and thus the number of cells in the grid-directory increase. Because of this, the two-disk-access'

principle is assured such that only two secondary memory accesses are made and the request rate for all the indexed data is relatively the same. The benefit of a high request rate is paid for by the increase in space consumption.

One of the problems which make traditional index structures inefficient is that these indices treat every data the same. If entries referenced by index tuple are queried frequently, they will be treated similarly to entries that are rarely or never queried. This information gets little attention in traditional approaches. The index structure needs the same space as before although the space is little used or if it is only used to read, for example, 20% of the referenced tuple. No priority is given to the frequency of the access on the data.

Another issue with many performance bottlenecks in the use of traditional DBMS is to anticipate increasingly difficult queries, for example people use Google to research when an interesting problem arise or an event happens in the community. There will be more queries than the DBA can predict. As the index treats every data the same, its structure does not reflect enough change in data access.

Only the production of partial indices (section 2.3.2) can be used but it must know in advance which tuples will be frequently or will not be accessed. This differentiation is often not practical and not easy verify in every case. In the classic example of the table that manages customer orders, the current orders are referenced frequently, while old orders charged with little or no queries. In order to detect the importance of data through access frequency, [11] suggests a new solution approach. In place of analyzing the workload, the authors get data access information from a higher level by integrating the data model and process model of the IT systems. Afterwards the data access flow will be analyzed to find those data items which are accessed frequently and the index is created automatically for these frequent items. The experiment of the project in [11] shows this approach can effective improve the data access performance.

In the next section, it is investigated how a restriction on the size of the KDB-tree can be implemented dynamically towards the data-access distribution.

4.2 Design of a load-balanced KDB-Tree

The aim of developing a load-balanced KDB-tree is firstly, the reduction of storage space consumption while still maintaining the same performance in comparison to a fixed size fully balanced KDB-tree without a load-balancing mechanism and secondly a more efficient access to more frequently used data. Both goals can be achieved by means of an autonomous tuning. This

means that the balancing in the tree can not only align to the distribution of data as described in section 2.4 of the KDB-tree, but can also increase the access to specific data fields.

Analogous to the procedure of the load-balanced binary tree, the KDB-Tree sets a maximum size of index nodes where points are located in point page. The more data are in a point node, the longer it takes to find an entry. In an original KDB-tree with a larger number of records increases the memory requirements of the index node. If the tree decreases the number of index nodes it must then increase number of buckets in order to manage all the data.

The KDB-Tree consists of two components that affect its performance: index node and data node. Both depend on each other directly. If the complete index nodes reside in main memory (for example) the cost for searching data depends on the number of access on point page. To adapt the tuning mechanism we have to reconsider these components of the KDB-tree.

4.2.1 The structure point page

From now on a point page is referred to as a bucket. A KDB-tree, as described in 3.4.1, offers no possibility of limiting memory requirements. This always depends on the number of entries that a leaf node can hold. To meet the demand for storage space boundaries, the KDB-tree structure is proposed to extend the structure point page. A point page for this paper will be known as a "bucket". A bucket can point to another bucket as a descendant. All of these buckets together create a linked list to store the records (called a bucketlist).

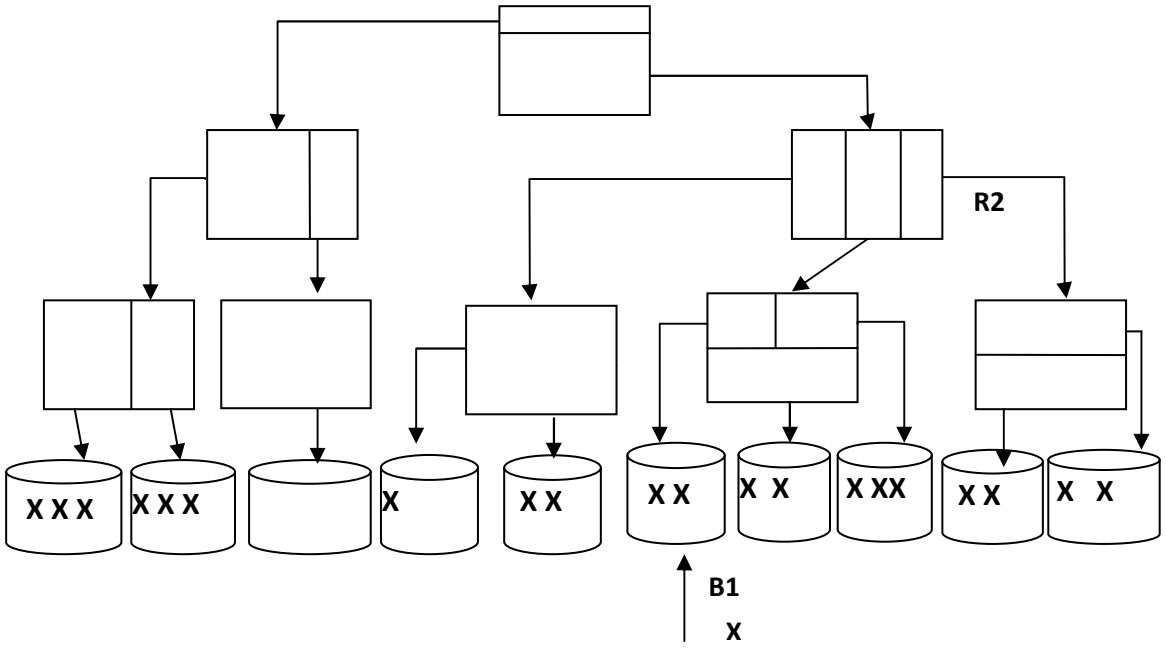

Figure 4.1: KDB-tree with fixed size of index node

The organizational form of an unordered list is selected to store the data inside a bucket. For another application a different organization form can be chosen. If the bucket is designed primarily to read data, we should store data in a selected order to increase the performance.

During the insertion of data, if a bucket is overflowed, a new bucket will generate and be linked to the original bucket to continue storing the inserted data. A bucket can have only one successor bucket.

We take the KDB-tree example from the previous chapter 2 (section 2.4) to apply a modification for a new structure. Remember that the KDB-tree is used to store 2-dimensional record; each bucket can contain a maximum of 3 entries. In the figure above, a KDB-tree is shown, which is filled almost completely. Inserting a new entry to bucket B1 for example would result in the splitting of parent region page R22 as well as index node R2. Inserting a record leads, in this case, to the consumption of more than 2 index nodes. Without exceeding the limit of 8 index nodes, this tree cannot receive any more entries. Inserting an entry would have to be rejected and the index would reflect the relation only partially. Both are exclusion criteria in the DBMS.

The generated overflow on the bucket must now be treated differently. Since the maximum number of index nodes has been reached, the only alternative is to create a new bucket and store the value there. We observe now the index node R22 which has bucket B1 as a leaf node.

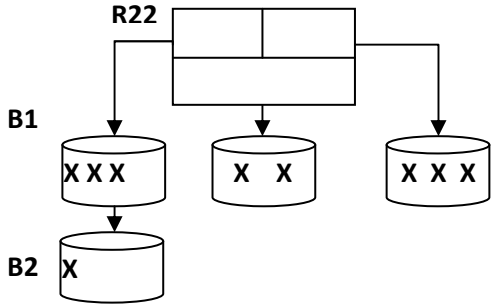

Figure 4.2: Insert record to an overflow bucket

A new bucket B2 is created and linked to bucket B1 to store the new entry. This type of linkage has the disadvantage that the record would have actually been on the first bucket to be found, but was moved to a next bucket. Therefore, search for a tuple in the common way needs more buckets than usual, which increase the I/O cost.

4.2.2 The structure index node

In order to adapt the new tuning concept on the KDB-tree, the index node must also change its structure. To ensure that the tree can expand or shrink index node depending on access frequency, the index node is modified so that it can have both point page and region page as

descendants at the same time. In the original KDB-tree the index nodes of the lowest level have only leaf nodes as point pages. Conversely in the load-balanced KDB-tree, the index node of a higher level can point to a bucket.

4.2.3 Statistics information management

In order to make a statement about the frequency of accesses, the index nodes and buckets must store the number of current requests. In this approach a distinction can be made between different types of accesses, each looking for a specific value in the tree.

- Selection - Finding a value
- Insert - Finding a value for locating the insert
- Delete - Finding a value for the deletion

This access information enables a differentiated statement on the use of index nodes and buckets. For example it represents exactly the areas of an index, where the insert operation most take place, combined with the selection access. It gives us more accurate on information which part of index should be the candidate for a reorganization process.

Another question needed to clarify is whether the statistical information should be stored in the index node and bucket or in a histogram. In a histogram the estimation of page accesses for a multidimensional record needs more study, because in KDB-tree the calculated unit is a region page with is much smaller than range value defined in the histogram.

4.2.4 Load-balanced KDB-tree

Assuming that the KDB-tree shown in Figure 4.3 has a limit of 8 index nodes and all the records are stored in a bucketlist, a query counter is implemented in every index node and bucket. A point query will increase the query counter in the index node and bucket by one as it goes through. By contrast, a range query request increases all index nodes that are involved in a range query on that path from root to bucket level by one(where the query must “go through” before reaching the bucket that holds queried record).

Assuming that the KDB-tree from figure 4.3 obtains 500 bucket accesses which are distributed among index nodes and buckets such as in the next figure 4.5, access statistics show that requests in the left subtree (140 accesses) clearly is lower than in the right subtree (360 accesses). The KDB-tree has a total 10 bucketlists

In principle the balancing of the tree works as follows: the root index node keeps track of all page accesses and the current number of bucketlist in the tree. The tree is considered balanced if it has:

$$balance_{KDB} = \frac{all\ page\ accessed}{total\ number\ of\ bucketlists}$$

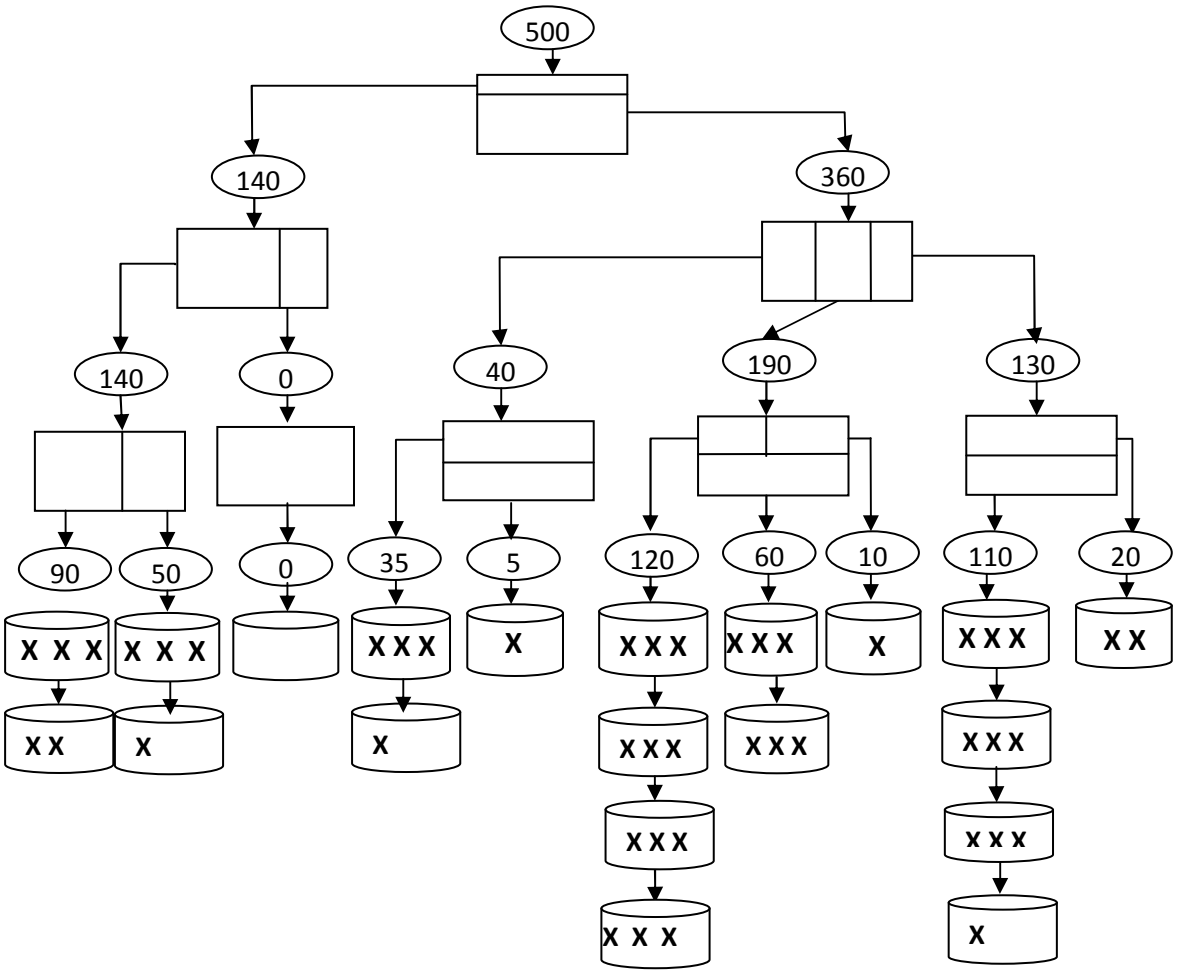

Figure 4.3: KDB-tree with page accesses statistics

During load-balancing reorganization, two basic methods are applied. Index nodes whose page access frequency are below $balance_{KDB}$, have to be removed from the tree and added to a pool of free nodes. The total page access of an index node is the cumulative page access of buckets that belongs to it. It is imperative that when an index node is removed, all index nodes of lower level along that index node must be also detached. It is necessary to find an index node or a group of index nodes whose total page access fulfills:

$$page\_accesses_{regionpage} < balance_{KDB}$$

We have: $$balance_{KDB} = \frac{500}{10} = 50$$

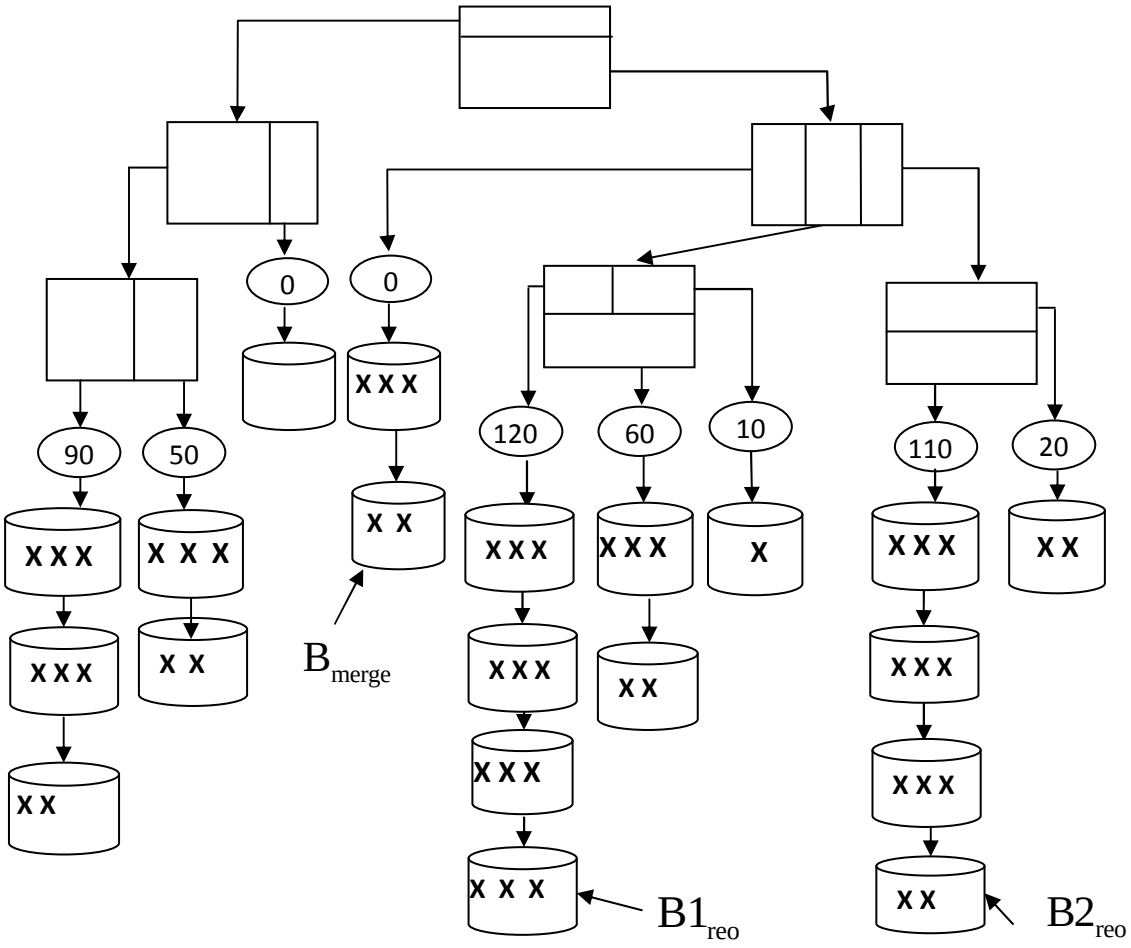

Figure 4.4: KDB-tree after removing an index node

Based on accesses statistics, we have two index nodes whose page accesses (0, 40 respectively) are smaller than $balance_{KDB}$ (50). We can remove these two nodes from the tree and add to a pool of free nodes for later use. The figure 4.4 shows the result of this remove-operation. Here the points in the buckets are now allocated in bucketlist B_{merge} and this bucketlist linked directly to the father region page of higher level. Here we can decide that the query counter on bucketlist B_{merge} should take zero value or the sum of cumulative bucket-accesses of bucketlists before merging. In prototype the query counter of B_{merge} is set to zero.

Instead of an index node the parent index node has now one new bucketlist as descendant at this position. The cost for searching data increases because more buckets have to be loaded into internal memory. One possibility to reduce the length of this bucket is that, if the father index node of this bucket still has free entries, than the newly created bucket will be split correspondingly to the free available entries. Figure 4.5 shows an example.

The bucketlist A links direct to index node R after reorganization. If the index node R can store a maximum of 4 subregions, that means a subregion can be added to this index node. We can partition subregion R1 into two sub spaces R11 and R12. The data must also be relocated correspondingly to new subregions and as result we have two bucketlists (A1and A2).

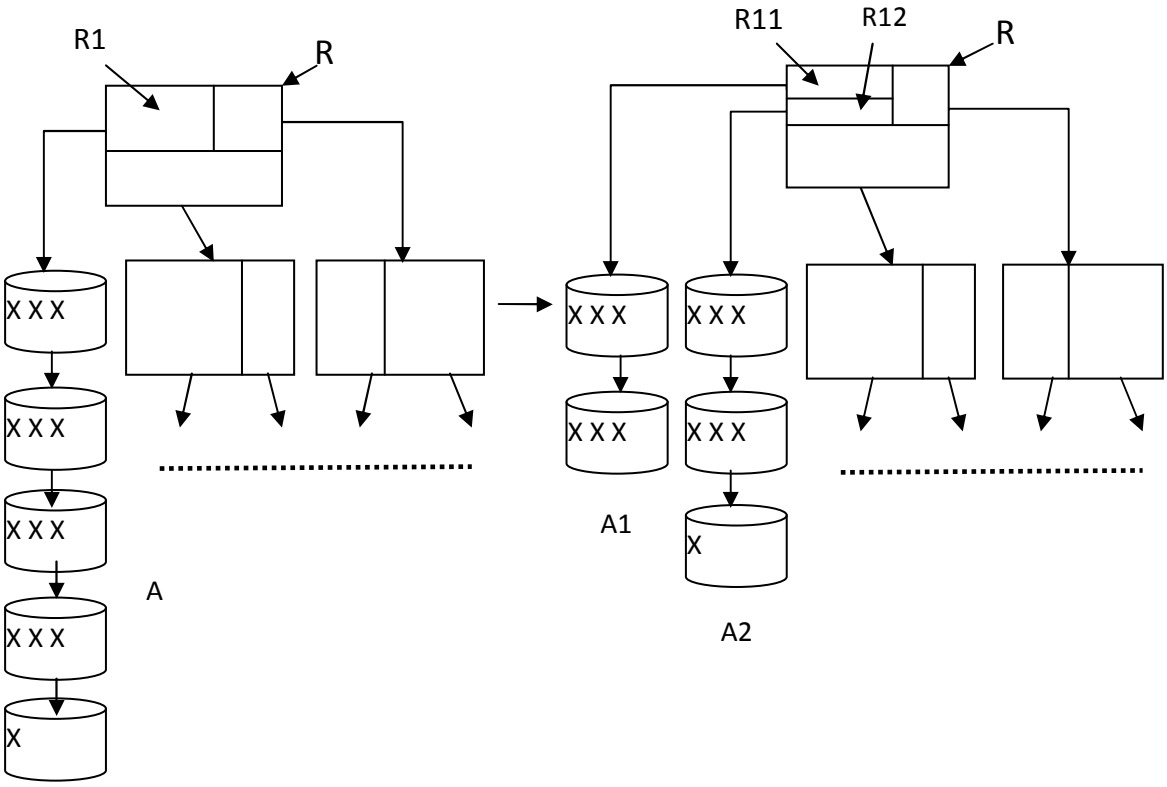

Figure 4.5: Reorganize bucketlist after merging bucket

Other buckets and index nodes that are not affected by removal operation remain unchanged including the query counter. The next phase of load-balancing is to use the free index node from the pool of free nodes to reorganize the bucketlist with high page accesses. With the aim of decreasing the search effort, a reduction of the length of the bucketlist in which requested data are frequently searched is desirable. To facilitate this, we have to address the bucketlist that has a high access and consider if the tree has available free index nodes to perform the restructure of this bucket-list. The following condition is:

$$number\_accesses_{bucketlist} > balance_{KDB}$$

As we can get from statistical accesses information which shows six bucketlists whose page accesses are much higher than $balance_{KDB}$. Here the bucketlists are sorted from highest to lowest page accesses. Since we have only two free index node, we select the bucketlists which have the highest page accesses and rebalance these bucketlist, here are bucketlist $B1_{reo}$, $B2_{reo}$ (figure 4.6).

Here the bucket is chosen to restructure and not the region page of the KDB-tree. The reason is that: firstly the index node in the KDB-Tree does not store the records. A cell or an interval in grid-file directory points only to one bucket-list; in contrast an index node at the lowest level of the KDB-Tree points to more bucketlists as its children. Secondly, as described in chapter 2 about the KDB-Tree, a split on index node can lead to a cascading split of another index node which leads to waste of free index nodes in the pool.

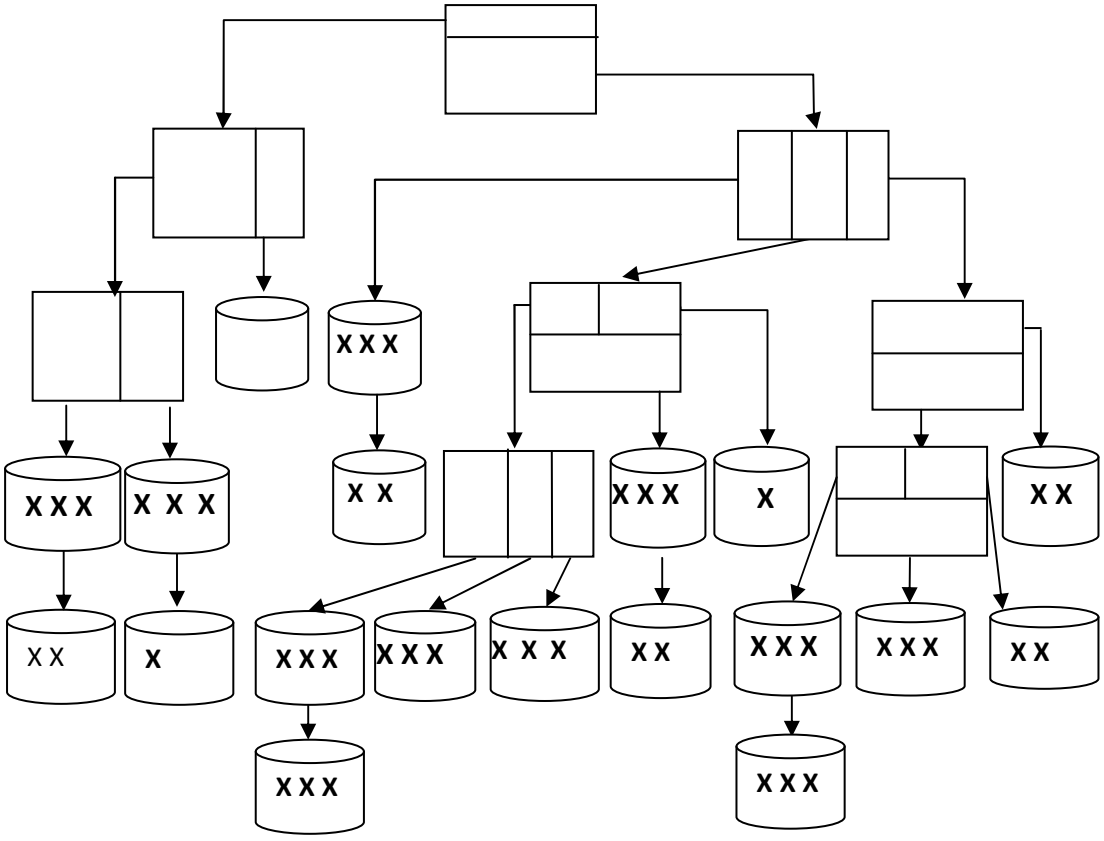

Figure 4.6: KDB-tree after reorganizing bucketlist

The other bucketlist within that index node does not get the effect. By this way, the length of the bucketlist with highest access will be reduced, and the search will be faster in the new created buckets

This example, despite the restrictions that the tree has one deep level, illustrates how the tree is staggered because of the different access frequency. The right part of the tree values is scaled

deeper, because the free index-nodes are used to restructure the buckets. Changing the frequency of access so that the left subtree would be preferable, this would be done in the same way on the left: Merge and Split if necessary.

4.2.5 Theoretical behavior of the load-balanced KDB-Tree

The self-tuning concept of a load-balanced KDB-Tree is implemented such that the index structure that can adjusts automatically to request samples. Part of the KDB-tree will be reduced if it gets lower request rate while another part will be expanded to adapt to the high frequency of accesses. To do this each query will be counted. The query counter is implemented in every index node.

Each time, when the KDB-tree reaches an amount of queries (for example after 1000 queries), The KDB-Tree will be reorganized towards the load-balance access mechanism to fit with this query workload. After this we can set all query counters in the index node and buckets to zero and waits for the next workload cycle. Alternative the query counter of new created bucketlist after merging (B_{merge}) has the total page accesses of all buckets belongs to it. and the bucketlist created after reorganizing of the index node (figure 4.6) take the value divided by the bucketlist before splitting ($B1_{reo}$ has 120 page accesses before reorganizing, after splitting each new bucketlist in this case has value 120/3=40 page accesses)

Inserting a new record in the load-balanced KDB-tree works as in the original KDB-Tree until it reaches the size limit. The new bucket will be created and linked to the overflowed bucket in order to store the new entry. The split operation on the bucket muss not occur, thus speed up the insert operation.

Deleting a record within a bucket that does not directly link to a father region page occurs in a simple way. The deletion of larger amounts of data is not considered in the evaluated Prototype where the merging operation of index node needs to be involved.

The adaptation of the KDB-Tree to the workload is triggered by the global request counter after given number of lookups. This leads to a regular balancing process and reorganization between bucket and index nodes, until it reaches the balance where all index nodes have balanced access equal to or bigger than $balance_{KDB}$. The search effort will increase accordingly, at least for part of the requests where multiple buckets are needed to search. During the adaptation of the KDB-Tree to the access distribution, this search effort should be decreased.

Chapter 5

Implementation

The following chapter describes the implementation of the load-balanced KDB-Tree developed in Chapter 4. It created also a test environment to investigate the behavior of KDB-Tree with different random values. The implementation was based on the original C++Classes developed by Beomseok Nam (The University of Maryland at College Park). It was rewritten in Java language. Upon this, the KDB-Tree was modified towards the concept of Kai-Uwe Sattler, Eike Schallehn and Ingolf Geist as described in [15].

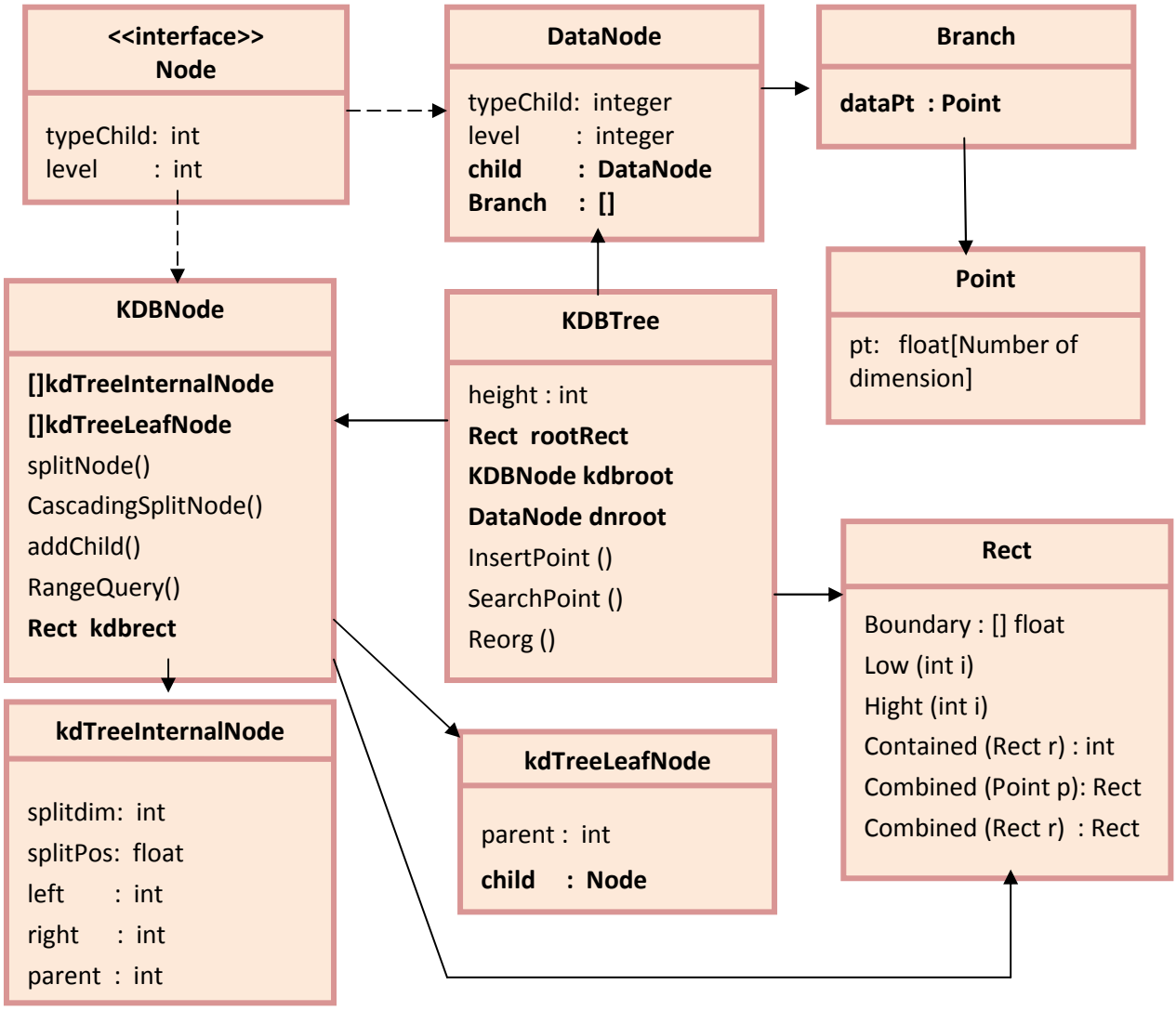

Figure 5.1: Class diagram of KDB-tree.

5.1 Limitation of the prototype

The implementation of the load-balanced KDB-Tree was realized according to the explanation in section 4.3. Since this is a prototype used for evaluation of the introduced concept, it is not necessary to implement all basic function of an original KDB-Tree. In the preparatory stage some conditions related to implementation are explained below.

Specification of data

The starting point for evaluation is a static database. Consequently, during runtime of the reorganization process none of inserting or deleting operations are performed. Therefore it will not perform the merge of point pages when these pages underflow. The number of records in the database is arbitrary. The record has three attribute and the value for each dimension was generated randomly using class `java.util.Random` from Java library (using `integer` data type).

5.2 Construction of the prototype and test environment

The implementation of both the prototype and testing environment was implemented using Java language. All Java classes needed for the implementation were in Package l`astkdbtree`. The important classes are:

- `DataNode.java` (plays role of bucket)
- `KDBTree.java`
- `KDBNode.java` (plays role of index node)

KDBNode.java consist of two classes: `kdTreeInternalNode.java` and `kdTreeLeaf-node.java` to perform the function of the KDB-Tree.

The DataNode.java plays role of the bucket, which is based on the concept in [15]. When a new point is inserted in a bucket that is full and the KDB-Tree has reached the fixed size, a new data node will be appended to this bucket (by using to methods `setForceBucket(true)` and `setChildNode(new DataNode())` in `Class DataNode.java`) to store new point. The first bucket in the bucketlist whose ancestor is a KDBNode will store static information about page accesses and number of queries.

As explained in section 2.4, the KDBNode play the role of a KD-Tree. It has `internal_Node` and `leaf_Node` where the `leaf_Node` has descendants as `KDBNode` or `DataNode`. In the prototype the data structure Array was used to implement the `internal_Node` and

`leaf_Node` structure. In this package, an interface Node is defined from which KDBNode and DataNode are derived.

The most relevant methods of the class for a KDB-Tree as well as the interaction between the classes are shown in Figure 5.1. At first the KDB-Tree has only one Datanode to accommodate entries. During the runtime as it is full of Data, it will be split into two new Datanode and a new KDBNode would be inserted.

The DataNode class in contrast to the KDBNode class is the place to store the data. When a Datanode is full, a domain (attribute of record) will be chosen for splitting and the median value along that dimension is calculated as a cut off value.

The following part explained some relevant aspects in detail. The pseudo-code gives an overview of the progress of each function.

Record search

The look up for a record is as described in Section 2.2.3. For the worst case, it has to search every bucket in the bucketlist to find the record. The first data node in the bucketlist will store the number of query and the total number of bucket access (the number of Datanode are invoked to find the record).

The search of record begins with the root of the KDB-tree, go through the index node (KDBNode) and then reach the point page (data node) where the data locates:

```
1 SearchPoint (Point p, Node root){
2  If(root is KDBNode){
3    Find the child of root that contains point p→ Node childroot;
4    If(childroot is DataNode){
5       //childroot is fist datanode
6       childroot.query++;
7       scan childroot and all of his children to find record ;
8      // here the query number and page access are added to
9    calculate the number of bucket accesses-> int n;
10       childroot.searchDatanode++=n;}
11       Else if(childroot is KDBNode) {
12            SearchPoint(p, childroot);
13                  }} }
```

Figure 5.2: SearchPoint procedure.

At line (9) the SearchPoint function returns the number of buckets that are invoked to find the record. If the record does not exist in the bucketlist, the returned value would be the length of that bucketlist.

Insert record to KDB-Tree

The function is used to insert new record in the KDB-Tree. To add the record, it must use the search method above to find Datanode where the record can be located. For this, the path from root to bucketlist is determined by comparing point's dimensions as well as its relative position to the hyperplane in KDBNode it goes through. While the splitting strategy for a data node is simple, it would be more complicated to split a KDBNode when its size is over-capacity. The partitioning of a KDBNode can involved a cascading split of many KDBNodes. More detail of the splitting method see the appendix. The pseudo code of the function is as follows.

```
InsertPoint (Node root, Point p) {
    // at beginning the kdbtree has only one DataNode also as root
If (root is DataNode){
    add p to root;
    If (root is full){
        split root → DataNode root1, DataNode root2;
        creat KDBNode  newkdbroot;
        add root1, root2 as child to newkdbroot;
                // the kdbtree has now the root as kdbnode
        root=newkdbroot;    }
If (root is kdbnode){
    SearchPoint(p,root)→ DataNode dnroot ;
 // searchpoint methode will return the DataNode that hold point p
    add point p to dnroot;
    if (dnroot is full AND max-kdbnode < limitKDBNode){
    find index node that point to dnroot→ (KDNode) father_kdbnode;
    split dnroot;
    add new internal_node and leaf_node  to father_kdbnode
 // more information on this step, sie the source code in Appendix
    add new kdbnode to the kdbtree ;
         }
          // modify towards the concept of Eike
    If (dnroot is full of AND max-kdbnode >= limitNode){
        dnroot.serForceBucket(true);
        append new DataNode to dnroot → dnroot.setChildNode (new
DataNode child_dnroot());
        add point p to child_dnroot;
     } } }
```

Figure 5.3: InsertPoint() procedure.

Remove KDBNode from the KDB-tree

This function is discussed earlier in section 4.3.4. It will be used to detect the KDBNode whose page accesses are smaller than the $balance_{KDB}$.

```
TransformKDBNodeToDataNode(KDBNode kdb){
   Find all the bucketlist belong to KDBNode kdb → List dnlist;
   For i:=1→dnlist.lengh
       dnlist(i) set child dnlist(i+1);
   Find KDBNode that is ancestor of node kdb → (father_kdb);
   Find the position where father_kdb point to kdb→ integer I;
   father_kdb.setChild(dnlist(0));
   put the free index node to the pool;
}
```

Figure 5. 4: Remove KDBNode procedure.

Reorganize a Bucketlist

This function tries to reorganize a bucketlist when that DataNode's page-access higher than the $balance_{KDB}$. It takes the free index node from pool to reorganize this bucketlist.

```
ReorganizeDatanode (bucketlist dn){
   Find the parent of DataNode dn →KDBNode kdbparent;
   creat new KDBNode newkdb;
   kdbparent set child newkdb;
   if the kdbnode not full of entries (see section 4.3.3) → value V;
   collect all the point belong to bucketlist →List pointlist
   divide the pointList into V bucket_list → Bucket_List
     For i:=1→ Bucket_list.length
         newkdb set child PageContainerList(i);
}
```

Figure 5.5: Reorganization a bucketlist procedure

Inform the $balance_{KDB}$

To calculate value of $balance_{KDB}$, we get the value from the root of KDB-Tree where all the queries must go through and all the page accesses gathered at time of reorganization.

For load-balancing method based on page accesses:

```
BalanceKDB(Node root){
  count all the page access to search the record in KDB-tree
  total_pageaccess;
   count all the bucketlist exist in KDB-Tree→  total_bucket_list;
    calculate the balance of tree
    → balance_KDB =total_pageaccess/total_bucketlist;
    return balance_KDB ;
}
```

Figure 5.6: Balance KDB-tree calculation procedure

Reorganization of the KDB-tree

This method as explained in chapter 4 is invoked after the KDB-tree reaches a number of query. For example the KDB-tree receives 20,000 queries and after 1,000 queries, the KDB-tree is reorganized.

```
1 class Test {
2   public static void main (String args[]){
3      // create the KDB-tree
4      for int i:=1→20000
5          Point p=new Point();
6          kdbtree.InsertPoint(p);
7          }
8      // the queryset has 20,000 query
9       List queryset=new List<QuerySet>;
10       For int i:=1→ queryset.length
11         if(i%2000==0){
12           kdbtree.reorg(kdbtree);
13         }
14}}
```

Figure 5.7: Invoke reorg() procedure.

The function *reorg()* from line 12 of figure 5.7 in KDBTree.java performs the reconstruction of KDB-Tree. It invokes all methods that were defined in previous steps. The method `TransformKDBNodeToDataNode()` muss be executed first. After that the method `ReorganisDataNode()` is called. As the result the KDB-tree will not be balanced anymore, the

height from the root of the tree to the leaf is not the same at every position, some parts of the tree will be deeper where they receives more queries than others.

```
Reorg(Node root){
    get the root of tree→ KDBNode root;
    balanceKDB(KDBNode root);
    find all kdbnode smaller than balance_KDB → List kdbnodeToDataNode;
    For int i:=1→ kdbnodeToDataNode.length
        TransformKDBNodeToDataNode(KDBNode kdb);
    counting the free available kdbnode after step above;
    find all bucketlist that higher than balance→List bucketlistToKDBnode;
    For int i:=1→ bucketlistToKDBnode.length
    Convert all this datanode to kdbnode;
}
```

Figure 5.8: Reorganization of KDB-tree procedure

Delete record from KDB-Tree

The "Delete" function is not here implemented, since its role has little relevance for the experimental purpose.

5.3 Source code

The complete source code in which the prototype is implemented is attached on a CD with this thesis.

Chapter 6

Evaluation

To assess the quality of the solution design in chapter 4, in this section a detailed evaluation will be performed. Because of the restriction in implementation, the prototype uses point query to analyze the output of KDB-tree after reorganization. The Prototype implemented in chapter 5 is tested with different query sets and distribution functions.

6.1 Test system and Java version

The evaluation was performed on an Intel Core 2 Duo with 2.26GHz and 2 GB Ram (Random-Access-Memory) main memory and the Windows Vista operating system.

Mainly as a runtime environment, the Java version 1.6.0_03 was used.

Initial Situation

Based on some request pattern (partial match query, range query or domain query) as described in chapter 1, the KDB-tree can use different methods for choosing the domain and split value along that domain to balance the tree. As it does not affect heavily on reorganization, the randomly selected domain method and the median as cut-off value are selected.

Because this solution approach cannot be compared to an original KDB-tree with full index nodes, the experimental design intends to test a fixed size KDB-Tree without the load- balancing with a load-balanced mechanism KDB-Tree and the original KDB-tree (developed by John T. Robinson). The fixed size here means that the KDB-tree has a limited number of index nodes. At the beginning the KDB-Tree is initiated which has no load-balancing mechanism, and the insert operation acts like an original KDB-Tree. After the KDB-tree has reached the bounded size (by observing the number of index nodes), the bucketlist structure will be used to store the data.

For the test, 20,000 records were generated and processed as insertions. An index node can contain a maximum of five subregions. A bucket can hold 5 records. The KDB-tree is a 3-dimensional-tree which means each record has three attributes (X, Y, and Z). In a series of tests during implementation an of original KDB-Tree whose size is not fixed, needs approximately 2000 index nodes and 4500 buckets to store all 20,000 records. To evaluate the load-balancing

mechanism, the KDB-tree will be tested with 500 index nodes, 750 index nodes, and 1000 index nodes variant.

A record inserted in KDB-tree has three attributes. Each attribute of the record receives `Integer` value generated randomly between 0 and 2000, which are uniformly distributed using `java.util.random` function in Java library.

To verify how the KDB-tree adapts to request a pattern, this experiment applies three different random distributions, which are referred to as uniform, column, and gauss. Because these distributions are applicable along three attributes of the record, a combination of all them leads to a total of 27 multi-dimensional distributions. Only a small part of these combinations above are taken into consideration for evaluation.

Evaluation Metric: The number of bucket accesses and the maximum index nodes are used as the basic metric for measuring the performance of the solution design.

6.2 Query distribution function

Uniform distribution: is a uniformly distributed probability distribution. This means that all values are generated with the same probability. In the ideal case, it would be a horizontal line. As available in J2SE, class `java.util.Random` is a uniform distribution function. For the experimental purpose the minimum value is 0, maximum value is 2000.

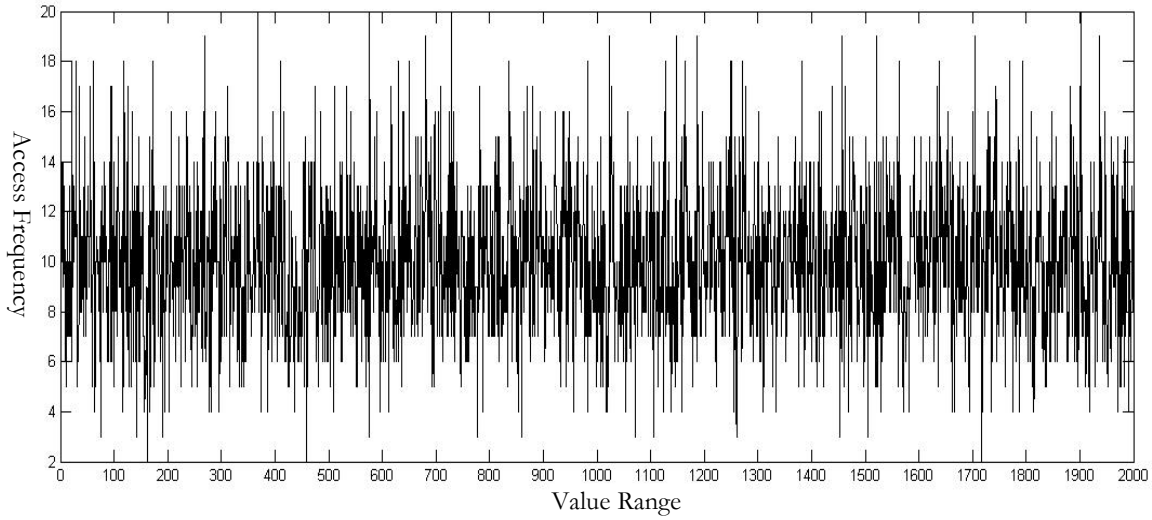

Figure 6.1: Uniform Distribution

Uniform Column distribution

Figure 6.1 shows the uniform column distribution. Like the uniform distribution, the value generated by the column function is also equally distributed. The difference lies in the definition of minimum and maximum. The uniform column distribution has a smaller range in comparison to a uniform distribution. For experimental purpose the minimum is defined at 500 and the maximum is 700, with approximately 10% of the range value of the uniform distribution above. The goal of this distribution is to distinguish two request patterns: while a uniform distribution presents the domain query pattern, the uniform column distribution reflects the range query pattern. The choice of minimum and maximum value here for column distribution is subjective and it can be another range value depending on the experimental purpose.

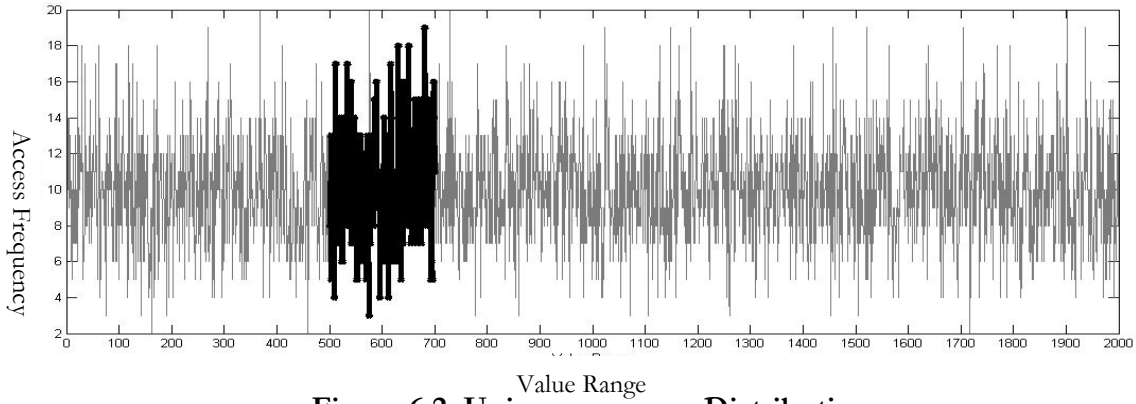

Figure 6.2: Uniform Column Distribution

Gauss Distribution

Gauss distribution (Figure 5-4) is a normal distribution, where the mean value has the highest probability distribution . Since the normal distribution is characteristic of many economic, natural, social, and other real world applications, it is useful to see how the prototype reacts with this query pattern. The standard deviation of gauss distribution here for testing purpose is defined subjectively.

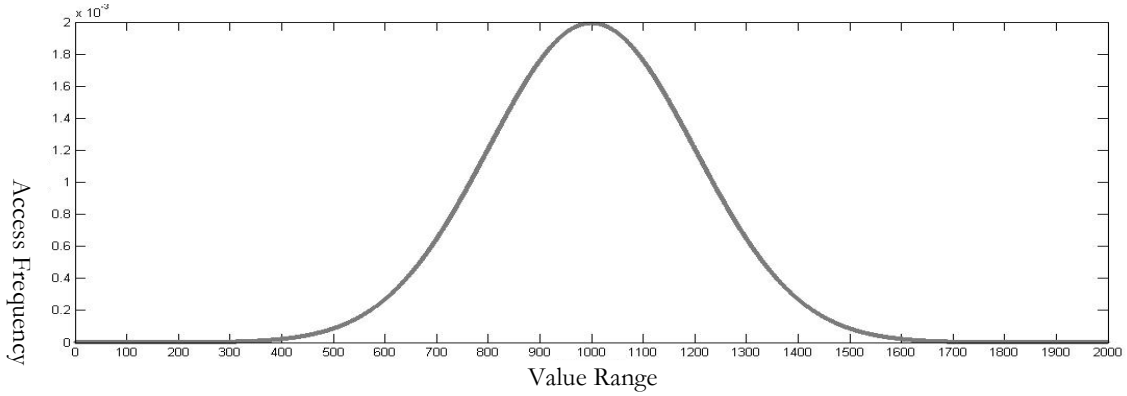

Figure 6. 3: Gauss Distribution (μ=1000, σ= 200)

Gauss Column Distribution

Like the uniform column distribution we also investigated the performance of KDB-tree where the value of the record takes on a gauss distribution (μ=1000, σ= 200). The range value is defined with a minimum value of 1000 and a maximum value of 1200. This form of distribution deals with queries where one attribute of a record has a high frequently access, such as how many items have been bought in a small period.

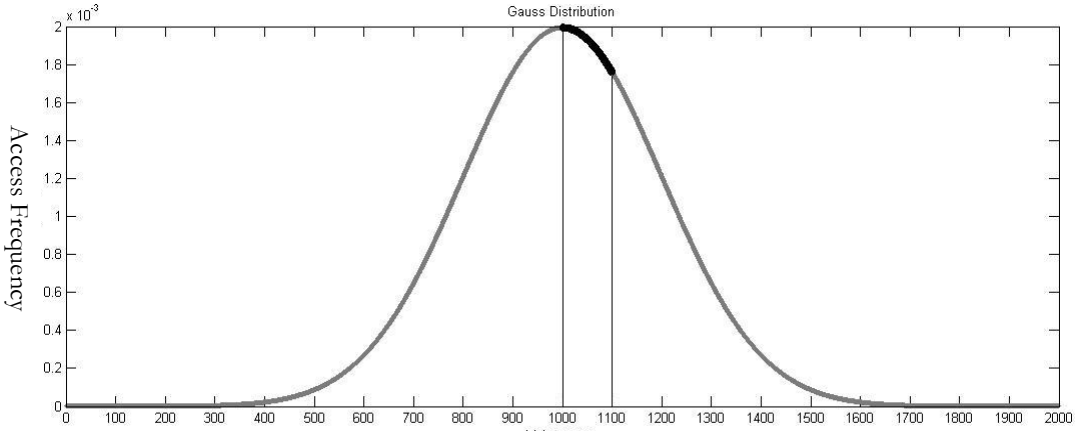

Figure 6.4: Gauss column distribution

The following experiments show the results of 20,000 lookup operations with and without balance mechanism. After each 1,000 lookup statistical information is gathered and evaluated before performing reorganization. Assuming that the total index pages are located in main memory, the main cost factor for retrieving data concentrates on the I/O disk access, which is a number of invocations of the buckets. The test is repeated ten times. After that, the average value was calculated to show effect of reorganization compared to fixed size KDB-Tree without access-balance mechanism.

Here in every test run, the choice of a 1000 request look up for reorganization was subjective. It means that the effect can be seen after 1000 requests. The access on the bucket will be hopefully faster from time to time after frequent reorganization. A higher frequency of reorganization can lead to a problem such that not enough index nodes will be released (be removed from the tree) and could decelerate the access speed as the KDB-Tree must be locked temporary to restructure. On the contrary lower frequency reorganization also decelerates the access speed because it does not respond fast enough to the new incoming request pattern. The question of, "after how many request should tree take the restructure" needs to study more in a productive application.

For each of these query patterns a workload of 20,000 queries is created. The following query patterns are executed in sequence (each domain takes one of distribution function as explained above):

- Uniform, uniform, uniform
- Gauss, gauss, gauss
- uniform column, uniform, uniform
- gauss column, uniform, uniform

Due to the redundancy of data, all results are not displayed. The complete test protocols and the implementation of as well as query distributions are on the enclosed disk. For an explanation of the evaluation, parts of the results are presented below. Here again a size of 500 index nodes is considered as maximum size of KDB-Tree for testing a load- balancing mechanism.

6.3 Experimental result

For every query set which corresponds to a distribution function the test shows the result of load-balance KDB-tree in comparison with fixed size KDB-tree that has no load-balance mechanism and original KDB-tree. After 10 trials an average of page accesses can be calculated. An analysis and conclusions are discussed directly after each result.

Because the number of bucket accesses are caused by queries that go through a bucketlist, the more a bucketlist receives queries the more buckets are involved to find the record. At this point it would be interesting to test the load-balanced mechanism based on number of queries. So instead of calculating the balance of the tree based on number of bucket accesses, we can modify the form by using another parameter:

$$balance_{KDB\_query} = \frac{all\ queries}{total\ number\ of\ bucketlists}$$

And change the condition into:

$$number\_query_{region\ page} < balance_{KDB\_query}$$

$$number\_query_{bucketlist} > balance_{KDB\_query}$$

For implementation we add a new function in KDBTree.java:

```
BalanceKDB_Query(Node root){
//the totalquery is determined by checking the query counter root of KDB-
//tree where all queries go through
 Count all the queries KDB-Tree→ total_query;
 Count all the bucketlist exist in KDB-Tree→   total_bucketlist
 Calculate the balance of tree→balance_KDB_query =totalquery/ total_bucket_list;
    return  balance_KDB_query ;  }
```

Figure 6.5: Balance KDB-tree based query calculation

On the next section the table shows the results of two load-balancing mechanisms in comparison to the fixed size KDB-tree without load-balanced and original KDB-tree (John T. Robinson). In the test the balance mechanism based on page access is referred to as PAB and the balance based on query access as QAB.

6.3.1 Uniform, uniform, uniform

$0 \leq X \leq 2000;\quad 0 \leq Y \leq 2000;\quad 0 \leq Z \leq 2000$

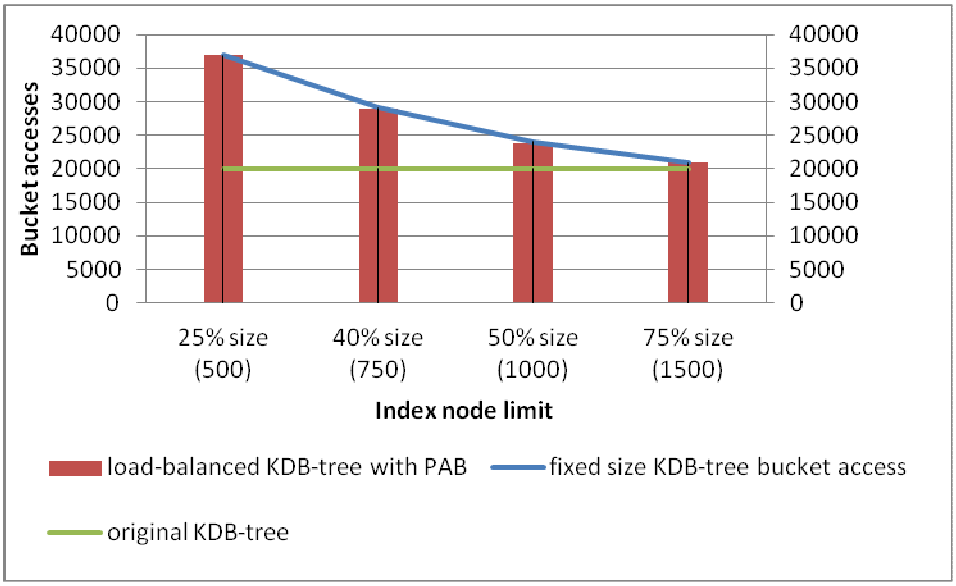

Figure 6.6: Bucket accesses of load-balanced KDB-tree with PAB

The figure exhibits the number of bucket accesses in two cases with PAB and QAB in comparison with a fixed size KDB-tree and original KDB-tree**.** As seen from the diagram above the PAB and QAB does not show a big effect on reducing the number of bucket access, and the index nodes used after reorganization remain as before (figure 6.8). It can be explained that all records distribute fast equally on every bucketlist, and only few index nodes need to be

reorganized. For uniform distribution of query (the data in database is also uniform distributed), the reorganization based load-balanced cannot provide profit.

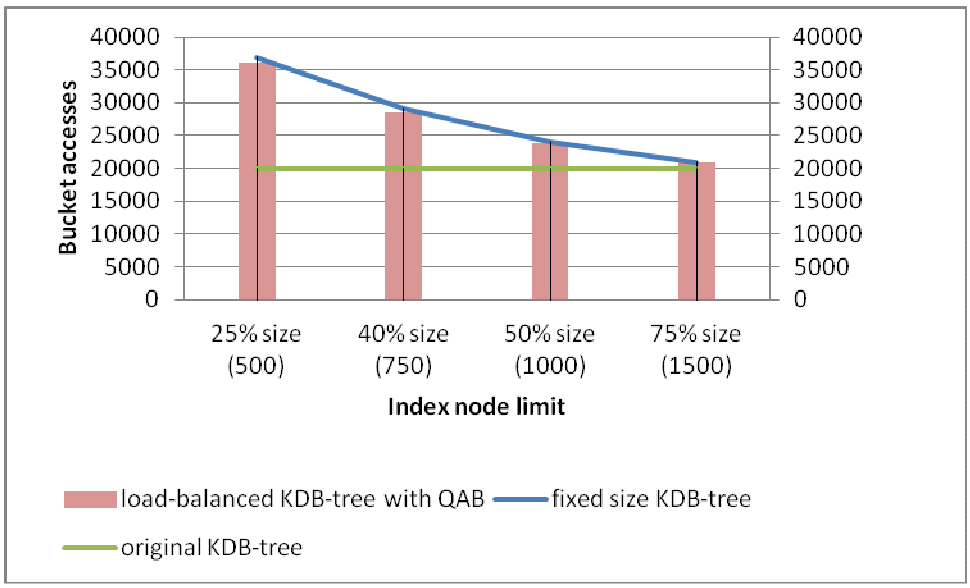

Figure 6.7: Bucket accesses of load-balanced KDB-tree with QAB

As we can see from figure 6.6 and figure 6.7, the load-balanced KDB-tree with 50% size of index nodes (compared with original KDB-tree) still shows a good performance in reducing the bucket accesses. This effect is not the result of load-balanced mechanism, instead it can be explained that the index nodes at the lowest level of load-balanced KDB-tree are full of entries, and all of bucketlists are full of records. The storage utility as a consequent is maximized

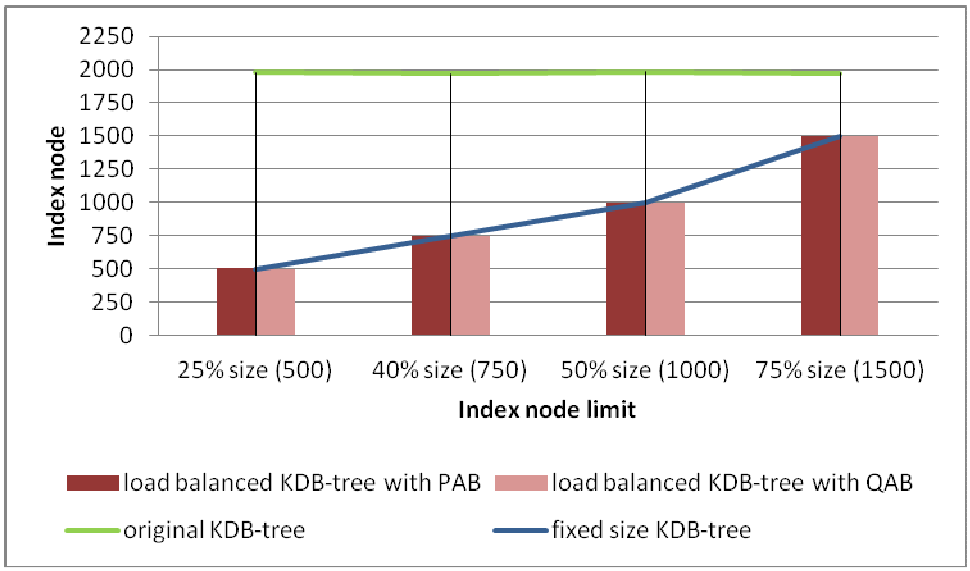

Figure 6.8: Index nodes used along index node limit

6.3.2 Uniform column, uniform, uniform

$$500 \leq X \leq 700; \quad 0 \leq Y \leq 2000; \quad 0 \leq Z \leq 2000$$

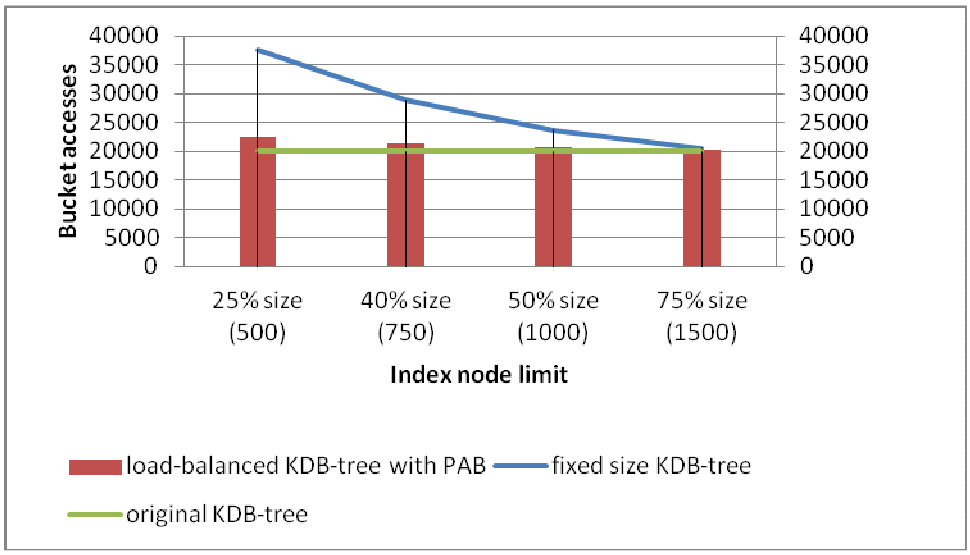

Figure 6.9: Bucket accesses of load-balanced KDB-tree with PAB

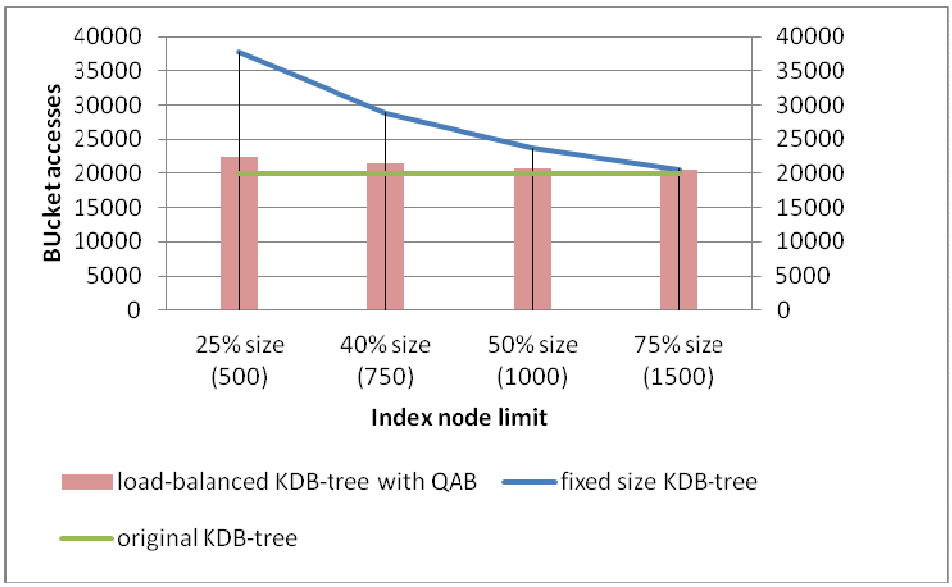

Figure 6.10: Bucket accesses of load-balanced KDB-tree with QAB

In this test, an attribute of a record receives query distribution (uniform column) in the range value from 500-700 (500≤ attribute_value≤700). It shows a clear difference between original KDB-tree and load-balance KDB-Tree. Both PAB and QAB have a better output in decreasing the number of bucket accesses. With the size limit at 25% (500 index node limit before reorganization), the amount of bucket access after balancing is approximately 10% higher in comparison with the original KDB-tree.

In addition, both PAB and QAB after reorganization consume less storage (index node) than the fixed size KDB-tree without load-balanced tuning. While the storage needed for original KDB-tree remains approximately near at 2000 index nodes, the bigger the limit size of load balanced KDB-tree before reorganization is (displayed in diagram 6.6 as a red line of fixed size KDB-tree)

the lower storage is consumed from the KDB-tree after reorganization. The result from test for example shows that a load-balanced KDB-tree before reorganization has a limit of 500 index nodes, after balancing the number of index nodes are reduced at approximately 460, in contrast a load-balanced KDB-tree with restriction of 1500 index nodes consumes only about 260 index nodes after reorganization.

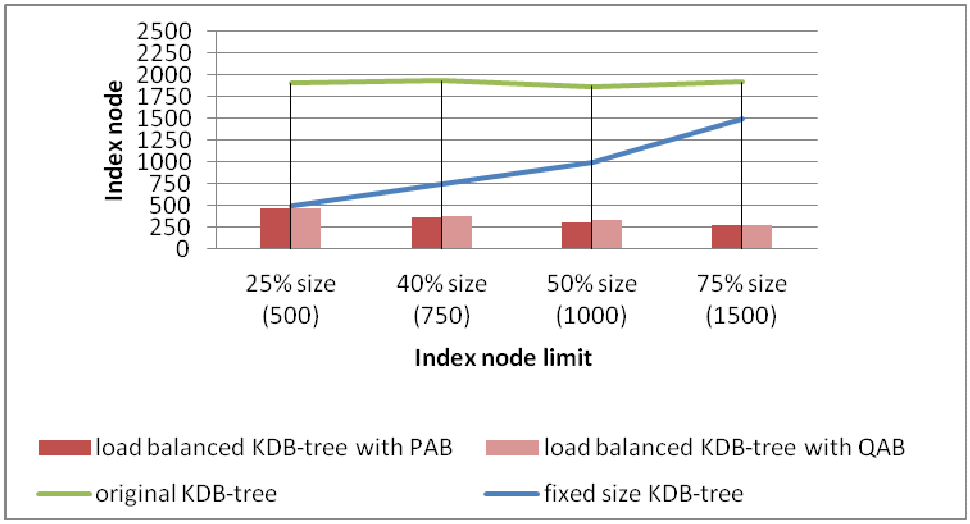

Figure 6.11: Index nodes used along index node limit

6.3.3 Gauss, uniform, uniform

$0 \leq X \leq 2000$ (μ=1000, σ= 200); $0 \leq Y \leq 2000$; $0 \leq Z \leq 2000$

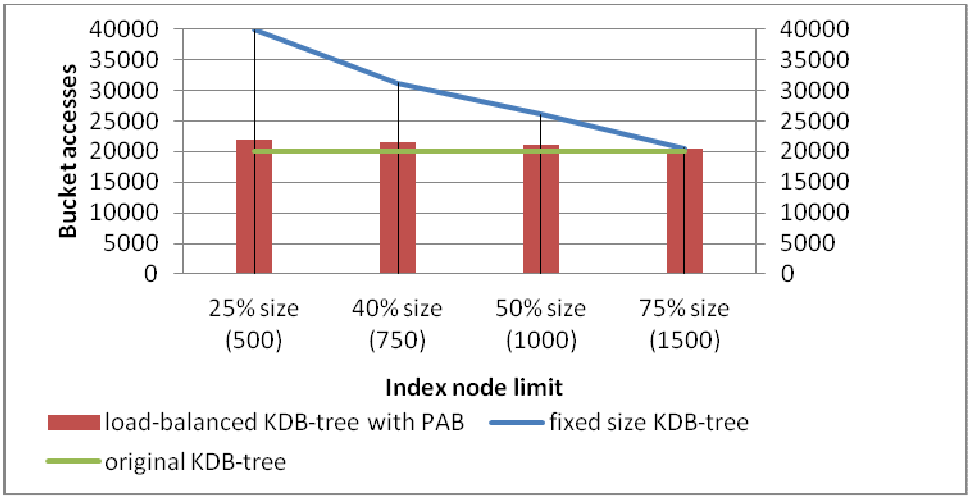

Figure 6.12: Bucket accesses of load-balanced KDB-tree with PAB

With this query distribution, the two PAB and QAB method show the same result. Both affect well on reducing the lookup effort and after reorganization the amount of index nodes are also reduced (figure 6.14).

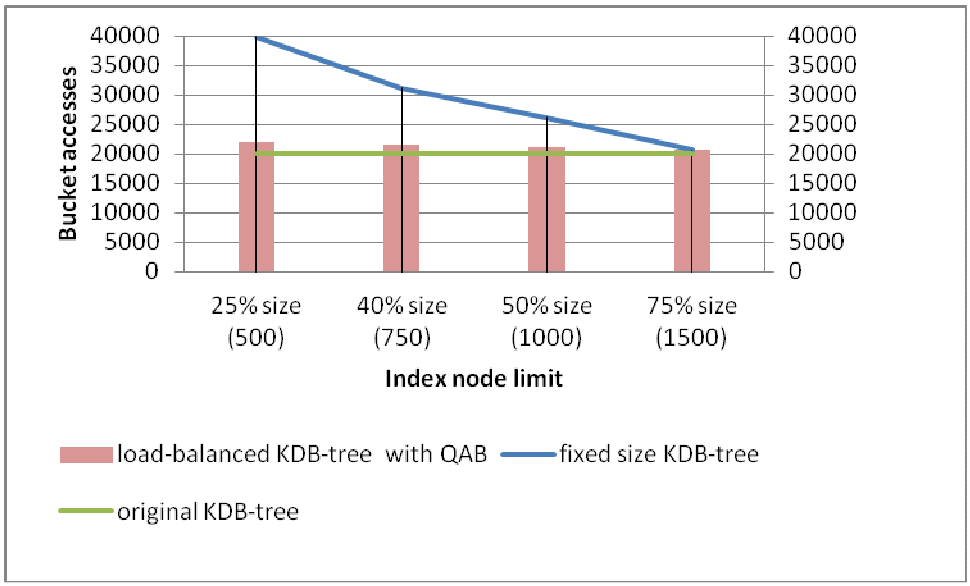

Figure 6.13: Bucket accesses of load-balanced KDB-tree with QAB

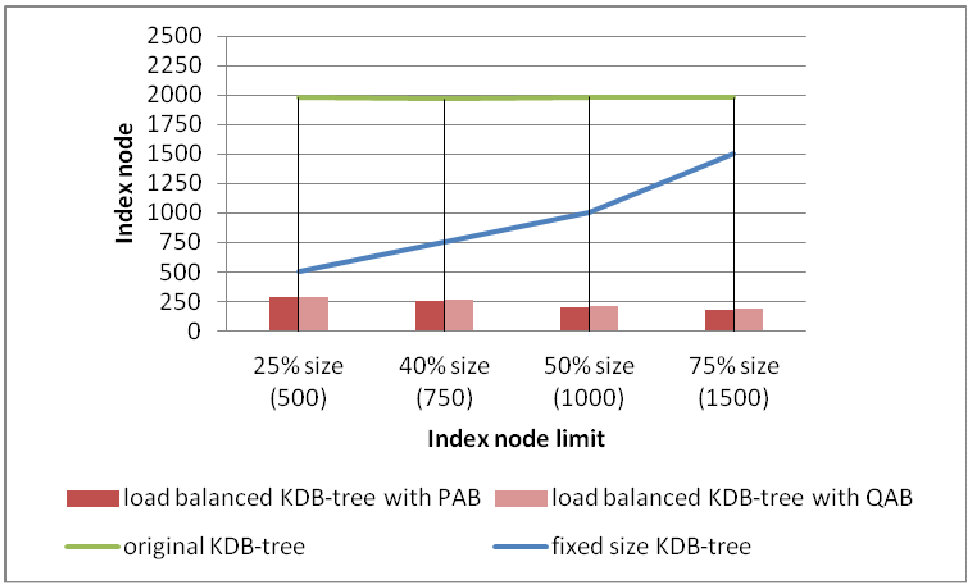

Figure 6.14: Index nodes used along index node limit.

Here at the size limit of 25% (500 index node limit before reorganization), the amount of index nodes after balancing is reduced on fast 50% while the bucket accesses approximately 5% higher than the original KDB-tree.

6.3.4 Gauss column, uniform, uniform

$1000 \leq X \leq 1200$ (μ=1000, σ= 200); $0 \leq Y \leq 2000$; $0 \leq Z \leq 2000$

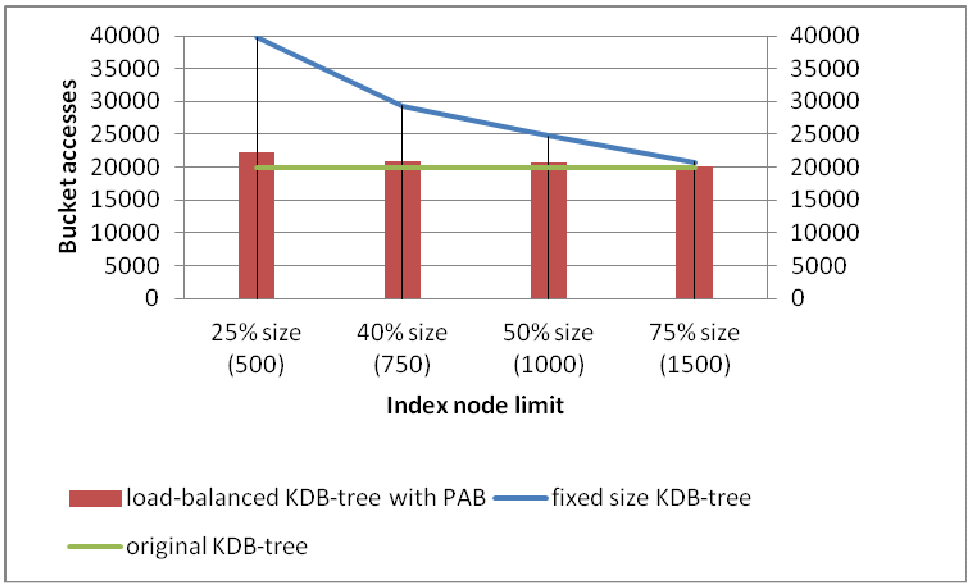

Figure 6.15: Bucket accesses of load-balanced KDB-tree with PAB

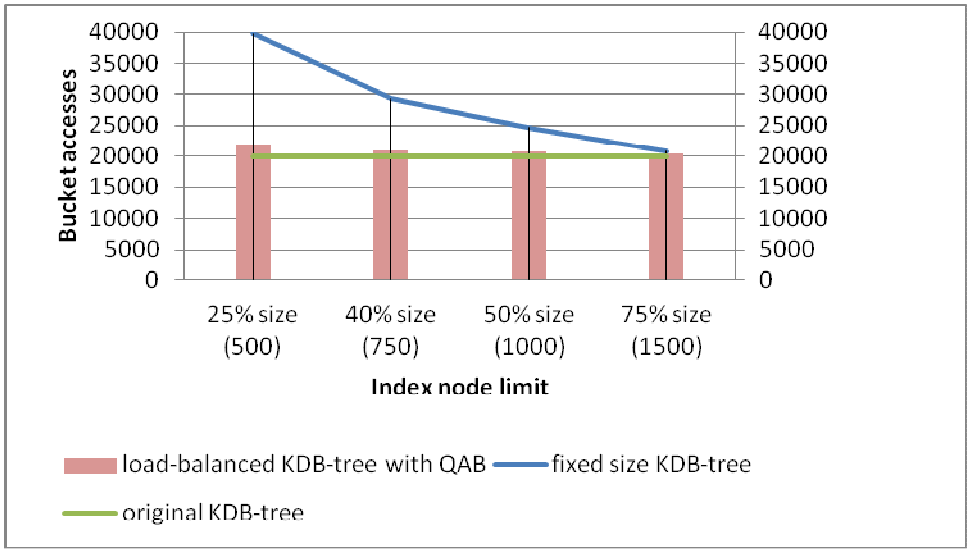

Figure 6.16: Bucket accesses of load-balanced KDB-tree with QAB

Here with gauss column distribution on one domain and uniform distribution along another domain, the PAB and QAB show better performance. Like the result in section 6.3.2 the lookup effort in both cases needs fewer index nodes and has fewer number of bucket accesses (versus fixed size KDB-tree). In comparison with original KDB-tree for example from the test result: at 25% size, the load-balanced KDB-tree needs about 14% less of index nodes and has the number of bucket accesses approximately 10% higher.

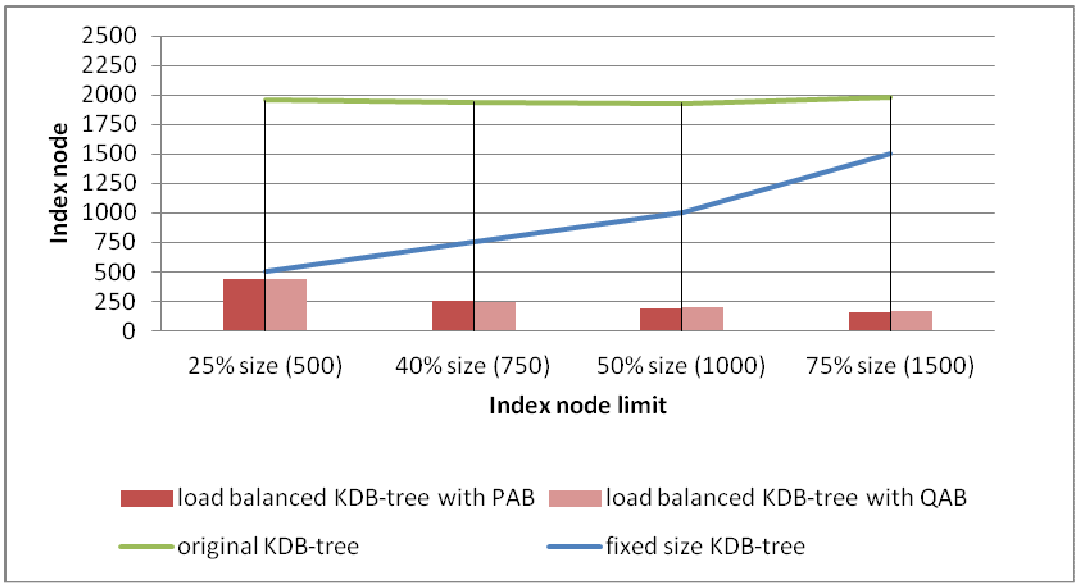

Figure 6.17: Index nodes used along index node limit

Like gauss distribution, the gauss column distribution also shows good performance. The query set requests data on restricted range value instead of spreading access to all record in database. Thus the load-balanced KDB-tree can detect which part of data has a higher access frequency and quickly react. As results the bucketlists where the highest accessed data are located, are reorganized and the index nodes which bounds the lower accessed data can be detached from the KDB-tree and added to pool of free node.

In summary we can say that with the reorganization mechanism above, the KDB-Tree reorganizes in general well within the range value of query patterns. With uniform column distribution and gauss column distribution the self-tuning concept of load-balancing shows the best effect in comparison to another distribution function. Another test that is not listed here also proved this result (more result can be read in the appendix). As the initial purpose of the original KDB-tree supports the range query, the adaption of the self-tuning concept with the load-balanced mechanism into the KDB-tree still proves its strength.

Chapter 7

Conclusion

The paper investigated how the KDB-tree introduced by John T. Robinson in [42] can be further developed with the new self-tuning concept of load-balanced access proposed in [15]. By measuring the importance of data which is based on the access frequency, the new concept enables efficient access to heavily requested data while the size for an index structure is limit. In order to adapt the self-tuning idea, the traditional structure (region page, and point page) of the KDB-tree must also be modified. As result the KDB-tree has an access-balanced structure instead of data-balanced structure.

The theoretic principal of a self-tuning concept was implemented and realized in a prototype. To evaluate the results of the implementation, the prototype was tested with different query distribution. Here the value spreading plays a crucial role: the smaller the range value requested in query for record is the more benefit the index provides (in case of uniform column and gauss column distribution). In summary we can say that the access-balanced KDB-tree has a good result with respect to the number of bucket accesses. In comparison to the original KDB-tree the number of bucket accesses converges for more often used data near to the optimum while at the same time the size of index structure is heavily restricted.

7.1 Future work

Although the first results achieved from evaluation showed that the self-tuning concept can provide a good solution to effectively manage the load-access KDB-tree structure, we must still cope with several tasks in the future.

First, the profit of the load-balanced self-tuning on a KDB-tree must be analyzed and verified for real world conditions. The prototype is so far executed exclusively with static databases and therefore it needs to be clarified whether the concept with insertion and deletions of records also works. It is necessary to examine how the prototype operates with different workload situations in a productive database system.

Second, as explained in previous chapters the number of bucket accesses depends on the size of KDB-tree index. The output of evaluation exhibits that the bucket accesses for KDB-trees with the load-balancing concept can be reduced to the same level as the original KDB-tree for a specific query pattern. Thus a question is how the size of access-balanced KDB-tree for a given data volume can be determined so that the number of bucket accesses can be decreased minimum. In a limitation of memory storage (knapsack problem), it would be useful to facilitate index selection.

Third, the implemented prototype stores the statistical information used for load-balanced reorganization on an index node and data node. It would be interesting to investigate how this information can be collected and evaluated by using histograms. For that purpose, the histogram for multidimensional index must also have the self-tuning ability in order to scale automatically to data-access distribution and not on the distribution of data itself.

Bibliography

[1] **Abraham Silberschatz, Henry F. Korth, S. Sudarshan:** Database System Concepts [Book]. - [s.l.] : McGraw-Hill, 2011. – 6

[2] **Keng Siau:** Advanced Topics in Database Research: 5

[3] **Ashraf Aboulnaga, Surajit Chaudhuri:** Self-tuning Histograms: Building Histograms Without Looking at Data [Conference] // ACM SIGMOD international conference on Management of data. - New York, NY, USA : [s.n.], 1999. - Vol. 28. issue 2.

[4] **ITL Education Solutions Limited**: Introduction to Database Systems

[5] **Bentley Jon Louis:** Multidimensional Binary Search Trees used for associatve searching [Journal]. - Stanford University : ACM, 1975. - Vol. 18. - 9.

[6] **Bentley, J. L. and J. H. Friedman (1979):** Data structures for range searching. ACM Computing Surveys 11 (4), 397-409

[7] **Comer, D. 1979**: Ubiquitous b-tree. ACM Comput. Surv. 11, 2, 121–137

[8] **Thomas M. Connolly, Carolyn E. Begg:** Database Systems: A Practical Approach to Design, Implementation, and Management [Book]. - [s.l.] : Pearson Education Limited, 2005. - 4.

[9] **Dennis Shasha Philippe Bonnet** Database Tuning: Principles, Experiments, and Trouble Shooting Techniques [Book]. - [s.l.] : Morgan Kaufmann, 2003.

[10] **Fabien De Marchi Mohand-Said Hacid** [Online]: Some remarks on self-tuning logical database design. - Univ. Claude Bernard Lyon 1. - 05 04, 2012. - http://liris.cnrs.fr/-Documents/Liris-1549.pdf.

[11] **Fu Xianghua Feng Boqin, Wang Xiaoming, Wang Zhiqiang** [Online]: Index Optimization for Database Based on Data Access Flow. - 05 07, 2012. - http://en.cnki.com.cn/Article_en/CJFDTotal-JSJC200712035.htm.

[12] **Gerhard Weikum, Axel Moenkeberg, Christof Hasse, Peter Zabback** Self-tuning Database Technology and Information Services:from Wishful Thinking to Viable Engineering [Conference] // Proceedings of the 28th VLDB Conference. - Hong Kong, China : [s.n.], 2002.

[13] **Gunter Saake Andreas Heuer** Datenbanken: Implementierungstechniken [Book]. - [s.l.] : MITP-Verlag Bonn, 1999.

[14] **Ivo Jimenez∗ Jeff LeFevre Neoklis Polyzotis, Huascar Sanchez,Karl Schnaitter**

[Online] = Benchmarking Online Index-Tuning Algorithms. - 05 04, 2012. - http://sites.computer.org/debull/A11dec/online.pdf.

[15] **Kai-Uwe Sattler Eike Schallehn, Ingolf Geist** Towards Indexing Schemes for Self-Tuning DBMS [Conference] // Data Engineering Workshops. 21st International Conference on. - 2005.

[16] **Kai-Uwe Sattler Ingolf Geis,t Eike Schallehn** QUIET: Continuous Query-driven Index Tuning [Conference]. - Magdeburg : 29th VLDB Conference, 2003.

[17] **Karl Schnaitter Serge Abiteboul, Tova Milo, Neoklis Polyzotis** On-Line Index Selection for Shifting Workloads [Online]. - 05 04, 2012. - http://users.soe.ucsc.edu/~karlsch/pubs/colt-paper.pdf.

[18] **Karl Schnaitter, Neoklis Polyzotis**: A Benchmark For Online Index Selection

[19] **Martin Lühring Kai-Uwe Sattler, Eike Schallehn, Karsten Schmidt** [Online]: Autonomes Index Tuning –DBMS-integrierte Verwaltung von Soft Indexen. - 05 04, 2012. -http://wwwiti.cs.unimagdeburg.de/iti_db/publikationen/ps/07/-LueSatSchSch07BTW.pdf.

[20] **Mensing Jan** Master- und Diplomarbeiten [Online]: Das Grid-File als zugriffsbalancierte mehrdimensionale Indexstruktur // Arbeitgruppe Dataenbank. - 03 18, 2011. - 05 04, 2012. - http://wwwiti.cs.uni -magdeburg.de/iti_db/-publikationen/ps/auto/thesisMensing.pdf.

[21] **Michael J. Corey, Michael Abbey, Daniel J. Dechichio** Tuning Oracle [Book]. - [s.l.] : Oracle Press, 1994.

[22] **Michael L. Rupley Jr.** [Online] = Introduction to Query Processing and Optimization. - 05 20, 2012. - http://www.cs.iusb.edu/technical_reports/TR-20080105-1.pdf.

[23] **Mittra Sitansu S.** Database Performance Tuning and [Book]. - [s.l.] : Springer Verlag, 2002.

[24] M**. Lühring, K.-U. Sattler, E. Schallehn, K. Schmidt**: Autonomes Index Tuning – DBMS integrierte Verwaltung von Soft Indexen; BTW 2007 S. 152-171

[25] **Mundt Sebastian** Master- und Diplomarbeiten [Online] = Last-balancierte Indexstrukturen // Arbeitgruppe Datenbank. - 29. 2 2008. - 4. 5 2012. - http://wwwiti.cs.uni-magdeburg.de/iti_db/publikationen/ps/auto/thesisMundt.pdf.

[26] **Nicolas Bruno, Surajit Chaudhuri** To Tune or not to Tune? A Lightweight Physical Design Alerter [Conference] // VLDB Endowment. - Seoul, Korea : [s.n.], September 1215,.

[27] **Nicolas Bruno, Surajit Chaudhuri** Automatic Physical Database Tuning:. - Baltimore : [s.n.], 6 2005.

[28] **Patrick Valduriez, Georges Gardarin:** Join and Semijoin Algorithms for a Multiprocessor Database Machine. ACM Trans. Database Syst. 9(1): 133-161 (1984)

[29] Piotr Kołaczkowski: Compressing Very Large Database Workloads for Continuous Online Index Selection. Copyright © 2008, Springer Berlin / Heidelberg

[30] **Polyzotis Neoklis** Self-Tuning Database Systems [Online]. - 05 04, 2012. - http://institute.lanl.gov/isti/issdm/projects/2010_CRP-Polyzotis-TBD-R.pdf.

[31] **Raghunath Nambiar, Meikel Poess** Performance Evaluation,Measurement and Characterization of Complex Systems [Conference]. - Singapore : Springer, 2010. - Second TPC Technology Conference, TPCTC 2010.

[32] **Sanjay Agrawal, Nicolas Bruno, Surajit Chaudhuri, Vivek Narasayya** AutoAdmin: Self-Tuning Database Systems Technology [Online]. - Microsoft. - 05 04, 2012. - http://research.microsoft.com/pubs/74152/deb.pdf.

[33] **Schallehn Dr.-Ing. Eike** [Online] = Database Tuning and Self-Tuning. - 05 20, 2012. - http://wwwiti.cs.uni-magdeburg.de/iti_db/lehre/advdb/tuning.pdf.

[34] **Shan Navathe, Nicolas Bruno:** Automated physical database design and tuning [Book]. - [s.l.] : CRC Press, 2011.

[35] **Stonebraker Michael:** The case for Partial Indexes. - University of California : [s.n.], 12 1989.

[36] **Stéphane Lopes, Fabien De Marchi, Jean-Marc Petit**: DBA Companion: A Tool for Logical Database Tuning

[37] **Stratos Idreos, Martin L. Kersten, Stefan Manegold** Database Cracking [Conference]. - Asilomar, California, USA : 3rd Biennial Conference on Innovative Data Systems Research, 01,2007.

[38] **Stratos Idreos, Stefan Manegold, Harumi Kuno,Goetz Graefe** Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores [Journal]. - Seattle, Washington : VLDB Endowment, 2011. - Vol. 4. - 9.

[39] **Su Chen Mario A.Nascimento, Beng Chin Ooi and Kian-Lee Tan:** Continuous Online Index Tuning in Moving Object [Journal]. - [s.l.] : ACM Transactions on Database Systems (TODS), July 2010 . - 3 : Vol. 35.

[40] **Surajit Chaudhuri, Gerhard Weikuma** Foundations of Automated Database Tuning [Conference]. - Seoul, Korea : VLDB '06, 2006.

[41] **Surajit Chaudhuri, Vivek Narasayya** Self-Tuning Database Systems: A Decade of Progress [Conference]. - Vienna, Austria : VLDB, 2007.

[42] **T.Robinson John:** The K-D-B-Tree: A Search Structure for large multidimensional dynamic Indexes. - Pittsburgh : [s.n.].

[43] **Tapio Lahdenmäki, Michael Leach:** Relational Database Index Design and the Optimizers [Book]. - Canada : John Wiley & Sons, 2005.

[44] R. Telford, The State of Autonomic Computing Today, IBM developerWorks 2004, http://www.ibm.com/developerworks/autonomic/library/ac-telford, 21.06.2007

[45] **Tarek Abdelzaher, Yixin Diao, Joseph L. Hellerstein,Chenyang Lu, and Xiaoyun Zhu** Introduction to Control Theory And Its Application to Computing Systems [Online] // SIGMETRICS. - 05 04, 2012. - http://www.cis.upenn.edu/~lee/09cis480/-papers/Sigmetrics.pdf.

[46] **Theo Härder, Erhard Rahm**: Datenbanksysteme: Konzepte und Techniken der Implementierung, 2. Aufl. 2001, Springer-Verlag

[47] **Tobin J. Lehman, Michael J. Carey:** A Study of Index Structures forMain Memory Database Management Systems [Conference]. - Kyoto : Proceedings of the Twelfth International, 1986.

[48] **Verma Ankit**:Enhanced Performance of Database by Automated Self-Tuned Systems [Conference] // IJCSMS International Journal of Computer Science & Management Studies. - May 2011. - Vols.11, Issue 01.

Appendix

The SearchPoint method for a KDBNode:

```
01 public int SearchPoint(Node n, Point p){
02         int hitcount=0;
03         if(n instanceof KDBNode){
04             KDBNode kdb=(KDBNode)n;
05             kdb.searchquery++;
06             Rect childNodeRect=new Rect();
07             int i=kdb.PickChild(p, kdb.kdbrect, childNodeRect);
08             if(kdb.getLeaf().get(i).getChild()!=null){
09             if(kdb.getLeaf().get(i).child.getType()==1){
10                  DataNode dn = (DataNode) kdb.getLeaf().get(i).child;
11                  assert(dn.sparent==1);
12                  dn.numberquery++;
13                  int a=dn.SearchPoint(p);
14                  dn.searchDataNode+=a;
15                  return a;
16             }
17             else if(kdb.getLeaf().get(i).child.getType()==2){
18                   hitcount=SearchPoint(kdb.getLeaf().get(i).child,p);
19             }
20             }else{System.out.println("record not exist ");}
21         }
22          return hitcount;
23     }
```

The SearchPoint method for a DataNode:

```
01 public int SearchPoint(Point p){
02     int hitcount=0;
03     int miscount=0;
04     ArrayList<DataNode> dlist=new ArrayList<DataNode>();
05     dlist.add(this);
06     if(this.haschild){
07         DataNode dn=this.getChildNode();
08         dlist.add(dn);
09         while(dn.haschild){
10          DataNode nd=dn.getChildNode();
11          dlist.add(nd);
12          dn=nd;
13        }
14     }
15      for(int i=0;i<dlist.size();i++){
16          DataNode dn=dlist.get(i);
17          miscount++;
18          if(dn.count>0){
19             for(int j=0;j<dn.count;j++){
20                Point temp=dn.branch[j].dataPtr;
21                int c=0;
22                for(int k=0;k<NUMDIMS;k++){
23
24                    if(temp.pt[k]==p.pt[k])
25                        c++;
```

```
26                }

27                if(c==NUMDIMS){
28                    return (i+1);
29                }}}   }
30      if(hitcount!=0){return hitcount;}
31      return miscount;
32    }
```

Insert a point to DataNode

```
01 public int AddChild(Branch b, DataNode new_node, split newsplit){
02        if(count<DATANODECARD){
03            branch[count]=b;
04            count++;
05        }
06         if (count >= DATANODECARD){ /* split it */
07      if(isForceBucket()==false ){
08
09         SplitNode(new_node, newsplit);
10                    return 1;
11             }
12             else if(isForceBucket()==true){
13                 if(this.haschild==false){
14                  DataNode newdn=new DataNode();
15                  newdn.sparent=0;
16                  this.setChildNode(newdn);
17                  this.setHaschild(true);
18                  newdn.setParent(this);
19                  newdn.setForceBucket(true);
20                 }
21                 else if(this.haschild=true){
22                       DataNode n=this.getChildNode();
23                       while(n.haschild==true){
24                             n=n.getChildNode();
25                       }
26                           n.AddChild(b, new_node, newsplit);
27                 }
28                 return 0;
29             }
30         }
31          return 0;
32     }
```

Split a DataNode method:

```
01  public void SplitNode(DataNode nn,  split newSplit){
02
03          DataNode replacementNode=new DataNode();
04          int target;
05          int i;
06          int best;
07          int minfill;
08
09          float f[] = new float[DATANODECARD];
10          assert(count == DATANODECARD);
11
```

```
12          // allocate replacement node and newnode
13          replacementNode.level=level;
14          nn.setLevel(level);
15
16 //  if(this.level==0){
17 //  java.util.Random random = new java.util.Random();
18 //  split_dim = random.nextInt(NUMDIMS); // split_dim.. randomly chosen
19          this.split_dim=(this.split_dim)%Infor.NUMDIMS;
20            for(i=0;i<count;i++){
21        f[i] = branch[i].dataPtr.pt[split_dim];
22            }
23
24            java.util.Arrays.sort(f);
25            if(f[0] == f[count-1]){
26        for(i=0;i<count;i++){
27       f[i] = branch[i].dataPtr.pt[split_dim];
28        }
29      java.util.Arrays.sort(f);
30            }
31            split_pos = f[count/2];
32          for(i=0;i<this.getCount();i++){
33      if(branch[i].dataPtr.pt[split_dim] <= split_pos){
34      //left..
35      target = replacementNode.getCount();
36      replacementNode.putBranch(target, branch[i]);
37      replacementNode.IncrCount();
38      }
39      else if(branch[i].dataPtr.pt[split_dim] > split_pos){
40      //right..
41      target = nn.getCount();
42          nn.putBranch(target, branch[i]);
43      nn.IncrCount();
44      }
45      else{
46      if( replacementNode.getCount() > nn.getCount()){
47          target = nn.getCount();
48              nn.putBranch(target, branch[i]);
49          nn.IncrCount();
50      }
51                  else {
52          target = replacementNode.getCount();
53              replacementNode.putBranch(target, branch[i]);
54          replacementNode.IncrCount();
55      }
56      }
57           }
58          this.setCount(0);
59          target=0;
60          newSplit.splitDim=this.split_dim;
61          newSplit.splitPos=split_pos;
62          this.split_dim+=1;
63          assert(replacementNode.Count() > 0);
64          assert(nn.getCount() > 0);
65          CopyFrom(replacementNode);
66          replacementNode=null; }
```

Insert a point to a KDB-tree. At the beginning the tree has only one DataNode

```
01  public void InsertPoint(Point p, DataNode Root, int Level){
02
03        KDBNode newroot = new KDBNode();
04
05        if(this.isRootDataNode==true){
06        int i;
07        Branch b=new Branch();
08        DataNode n2 = new DataNode();
09        split newSplit=new split();
10
11        int ret_val;
12        b.dataPtr = p;
13
14     ret_val=Root.AddChild(b,  n2, newSplit);
15        if(ret_val==1){
16                  /*grow a new root, & tree taller*/
17      newroot.setLevel( Root.getLevel() + 1);
18      newroot.CreatekdTree(Root /*left*/ , n2 /*right*/ , newSplit);
19        this.isRootDataNode=false;
20        this.KDBTreeRoot=newroot;
21        this.countKDBNode++;
22          }
23        }
24        else {
25            this.KDBTreeRoot=this.Insert(p, this.KDBTreeRoot, Level);
26        }
27     }
```

The method **`Insert()`** at line 25 from **`InsertPoint`**() method is listed below:

```
01  public KDBNode Insert(Point p, KDBNode Root, int Level){
02
03       // Node root = Root;
04        int level = Level;
05        KDBNode newnode=new KDBNode();
06        KDBNode newroot= new KDBNode();
07        Branch b;
08        split newSplit=new split();
09        assert(level >= 0 && level <= Root.getLevel());
10       /* root split */
11     if (Insert2(p, Root,  newnode, newSplit, level, rootRect)==1){
12        if(Root.getLevel() == 0){
13    // current root is a data node but new root would be a KDBNode
14      assert(Root.getType() == 1);
15      newroot.setLevel( Root.getLevel()+1);
16      newroot.CreatekdTree(Root /*left*/ , newnode /*right*/ ,newSplit);
17      Root=newroot;
18      height++;
19     }
20   else {// current root is ordinary node, new root is also ordinary root
21      assert(Root.getType() == 2);
22         this.countKDBNode++;
23      newroot.setLevel(Root.getLevel() + 1);
24      newroot.CreatekdTree(Root, newnode, newSplit);
```

```
25                  Root.countKDB=0;
26      height++;
27    }
28  return newroot;
29     }
30     else
31  return Root;
32 }
```

The method **`Insert2()`** at line 11 from **`Insert`** () method is listed below:

```
01 public int Insert2(Point p,  Node n, Node new_node,
02      split newsplit, int level, Rect parentNodeRect){
03
04         int i;
05         Branch b=new Branch();
06         KDBNode n2 = new KDBNode();
07
08         DataNode n22=new DataNode();
09         split lastsplit=new split();
10         Rect childNodeRect=new Rect();
11         Rect childNodeRectR;
12         Rect backup;
13         Rect coverRect;
14         int ret_val;
15         assert(level >= 0 && level <= n.getLevel());
16     // Still above level for insertion, go down tree recursively
17     //
18     if (n.getLevel() > level){
19
20  if (n.getType() == 2){  // minimum overlap node
21
22        KDBNode N=(KDBNode )n;
23        i = N.PickChild(p, parentNodeRect, childNodeRect);
24
25         if (Insert2(p,
N.getLeaf().get(i).child,(N.getLeaf().get(i).child.
26    getType()==1)? n22:n2, lastsplit, level, childNodeRect)==0){
27        return 0;
28        }
29        else{
30        ret_val=N.AddChild((N.getLeaf().get(i).child.getType()==1)?
n22:n2,
31        (KDBNode) new_node, i, lastsplit, newsplit);
32        if (ret_val==1) {
33                          node_split_info split_choice=new
node_split_info();
34                          N.GuaranteedAssessSplits(split_choice);
35
N.countKDBNodeAfterSplit(N.cn,split_choice.splitdim,split_choice.splitpos);
36                          this.countKDBNode+=N.cn.count;
37                          N.cn.setCount(0);
38        //for the prototype the limit of index node is here defined
39                          if(this.countKDBNode>1000){
40                              this.setLimitKDBNode(this.KDBTreeRoot);
```

```
41                          }
42                             N.SplitNode((KDBNode)new_node, newsplit);
43          assert((((KDBNode )new_node).numLeafNodes < KDBNODECARD) &&
44              (((KDBNode)new_node).numLeafNodes > 0));
45                      }
46                      return ret_val;
47        }
48  }
49     }
50     else if (n.getLevel() == level){
51  DataNode N=(DataNode )n;
52
53  b.dataPtr = p;
54  int ret= N.AddChild(b,  (DataNode)new_node, newsplit);
55         return ret;
56     }
57     /* Not supposed to happen */
58      return 0;
59 }
```

The **`AddChild()`** function from line 30 from previous function is listed below:

```
01 public int AddChild(Node newChildNode, KDBNode new_node, int
kdTreeLeafIndex,
02      split dataNodeSplit, split kdTreeSplit){
03
04        if(numLeafNodes<KDBNODECARD){
05        int parent=leaf.get(kdTreeLeafIndex).parent;
06
07        if(this.numInternalNodes==0){
08        }
09        if (this.internal.get(parent).left ==
(kdTreeLeafIndex+KDBNODECARD)){
10 //          System.out.println("numInteralNodes at moment:
"+numInternalNodes);
11     this.internal.get(parent).setLeft(this.numInternalNodes);
12        }
13        else if (this.internal.get(parent).right ==
(kdTreeLeafIndex+KDBNODECARD)){
14 //      System.out.println("numInteralNodes at moment:
"+numInternalNodes);
15     this.internal.get(parent).setRight(this.numInternalNodes);
16
17        }
18        else{}
19  //assert(FALSE);   // shouldnt happen
20
21        this.leaf.get(kdTreeLeafIndex).setParent(numInternalNodes) ;
22
23        this.internal.get(numInternalNodes).splitDim=
dataNodeSplit.splitDim;
24        this.internal.get(numInternalNodes).splitPos=
dataNodeSplit.splitPos;
25        // this assertion is true for point data
26
```

```
27
this.internal.get(this.numInternalNodes).setLeft(KDBNODECARD+kdTreeLeafInde
x);
28
this.internal.get(this.numInternalNodes).setRight(KDBNODECARD+numLeafNodes)
;
29        this.internal.get(this.numInternalNodes).setParent(parent);
30        this.leaf.get(this.numLeafNodes).setChild(newChildNode);
31
32        this.leaf.get(this.numLeafNodes).setParent(numInternalNodes) ;
33
34        this.numInternalNodes++;
35        this.numLeafNodes++;
36        }
37  if (this.numLeafNodes >= KDBNODECARD){  // the KDBnode is full,should
be split
```

The SplitNode() function is used to split a KDBNode:

```
01 public void SplitNode(KDBNode new_node,split newSplit){
02
03          int i;
04          int ptr;
05          node_split_info split_choice=new node_split_info();
06          KDBNode []kdbn1=new KDBNode[KDBNODECARD];  // left
07          KDBNode []kdbn2=new KDBNode[KDBNODECARD];  // right
08          for(int j=0;j<kdbn2.length;j++){
09              kdbn2[j]=new KDBNode();
10          }
11          DataNode []dn1=new DataNode[KDBNODECARD];   // left
12          DataNode []dn2=new DataNode[KDBNODECARD];   // right
13          for(int j=0;j<dn1.length;j++){
14            dn1[j]=new DataNode();
15        }
16          for(int j=0;j<dn2.length;j++){
17            dn2[j]=new DataNode();
18          }
19          Node []n1=new Node[KDBNODECARD];
20          Node []n2=new Node[KDBNODECARD];
21
22  assert(numInternalNodes == (KDBNODECARD-1)); //node is full, hence
splitting
23
24          GuaranteedAssessSplits(split_choice);
25
26          newSplit.splitDim=split_choice.splitdim;
27          newSplit.splitPos=split_choice.splitpos;
28
29          for(i=0;i<split_choice.numCascadingSplits;i++){
30       ptr =  split_choice.listOfSplittingChildren[i];
31
32       if( (this.getLeaf().get(ptr)).child.getType() == 2){
33      kdbn1[ptr] = (KDBNode) this.getLeaf().get(ptr).getChild();
34      kdbn1[ptr].CascadingSplitNode(kdbn2[ptr], split_choice.splitdim,
split_choice.splitpos);
```

```
35      n1 = kdbn1;
36                  n2 = kdbn2;
37       }
38       else if( this.getLeaf().get(ptr).child.getType() == 1){
39      dn1[ptr] = (DataNode) this.leaf.get(ptr).getChild();
40
41      dn1[ptr].CascadingSplitNode(dn2[ptr], split_choice.splitdim,
split_choice.splitpos);
42
43                  n1 = dn1;
44                  n2 = dn2;
45       }
46          }
47          SplitKDBNodekdTree(new_node, split_choice, n1, n2);
48   return;
49 }
```

The `reorg()` to reorganize the KDB-tree towards the load-balance access

```
01 //the limitnode is the number of index node that a KDB-tree can hold
02 public void reorg3(KDBNode kdb, int limitnode){
03        ArrayList<DataNode> dnlist=new ArrayList<DataNode>();
04
this.KDBTreeRoot.ListDataNodeNoneChildunderKDBNode(this.KDBTreeRoot,dnlist)
;
05     // gather statistic information before reorganisation
06        this.KDBTreeRoot.staticInfor();
07     // inform the balance of the tree
08        int balance=this.KDBTreeRoot.searchDataNode/dnlist.size();
09
10        this.countKDBNodeInTree=0;
11        this.countKDBNodeTree(this.KDBTreeRoot);
12  // remove all index node whose number of accesses smaller than balance
of the tree
13        transformAll3(kdb,balance);
14        this.countKDBNodeInTree=0;
15        this.countKDBNodeTree(this.KDBTreeRoot);
16
17     // count the residual index node after remove index node
18        int free=freenode-this.countKDBNodeInTree;
19
this.KDBTreeRoot.ListDataNodeNoneChildunderKDBNode(this.KDBTreeRoot,dnlist)
;
20
21  //the bucketlist are sorted based on accesses number, from Max-->Min
22        dnlist=this.sortDNAfterSDN3(dnlist,balance);
23        while(free>0){
24            if(dnlist.isEmpty())
25                break;
26            else {
27                DataNode dn=dnlist.remove(0);
28                if(dn.CountDataNodeChild(dn)>1){
29                    if(dn.searchDataNode>balance){
30                       KDBNode knew=new KDBNode();
31                       KDBNode parent=this.FindKDBNodeAboveDataNode(dn);
```

```
32                      parent.BoundedKDBNodeNew(dn, knew);
33                      free--;
34                  }
35              }
36          }
37       }
38      }
39       this.countKDBNodeInTree=0;
40    // set every query counter and access number to zero
41       this.SetTotalAccess(this.KDBTreeRoot);
42       this.SetRect(this.KDBTreeRoot);
43    }
```

```
39       this.countKDBNodeInTree=0;
40    // set every query counter and access number to zero
41       this.SetTotalAccess(this.KDBTreeRoot);
42       this.SetRect(this.KDBTreeRoot);
43    }
```

Email to: binhcongnguyenvn@yahoo.com for Sour code

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, den 16.06.2012

Nguyen Cong Binh