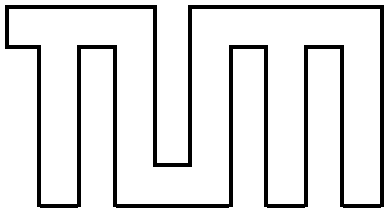
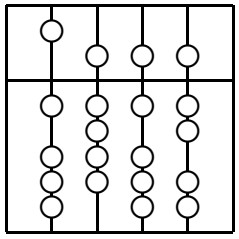




Institut für Informatik
der Technischen Universität München

Towards a general-purpose, multidimensional index:

Integration, Optimization, and Enhancement of UB-Trees

Frank Ramsak

Institut für Informatik
der Technischen Universität München

Towards a general-purpose, multidimensional index: Integration, Optimization, and Enhancement of UB-Trees

Frank Ramsak

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. A. Knoll
Prüfer der Dissertation:	
1.	Univ.-Prof. R. Bayer, Ph.D.
2.	Prof. T. Sellis, Ph.D., National Technical University of Athens/Griechenland

Die Dissertation wurde am 23.04.2002 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 17.06.2002 angenommen.

In memory of my father, Heinz Ramsak.
In place of the piano, Dad.

Acknowledgements

First of all, I want to thank my supervisor Prof. Bayer, Ph.D. for his advice, guidance, and support. Our discussions were an invaluable source of ideas and inspiration, not only for this thesis, but for my work as researcher in general.

I am particularly grateful to Volker Markl who always provided assistance, even after leaving the MISTRAL project. Our countless discussions, sometimes even in the middle of the night, contributed many ideas to this work. With his enthusiasm and dedication for research he is a shining example, giving motivation especially in the "periods of drought" one is inevitable going through during the Ph.D. studies.

I also found a valuable fellowship in my colleagues Robert Fenk, Roland Pieringer, and Martin Zirkel. Sharing the same passion for our project, they were always at hand when I needed help in theoretical or technical questions. The good team spirit in the MISTRAL project made the work not only exciting but also much fun.

I like to thank my former master students and interns, especially Ralph Acker, Stephan Merkel, and Anton Tichatschek, who contributed to this work by supporting the implementation of algorithms as well as performing some of the experiments.

Special thanks to the project partners of the MISTRAL project. AISoftw@re (Italy), European Commission, GfK Market Research (Germany), Globema (Poland), Hitachi (Japan), ICCS of NTU Athens (Greece), NEC (Japan), Microsoft Research (USA), SAP (Germany), Teijin Systems Technology (Japan), TransAction Software (Germany), and Unixfor (Greece) did not only provide funding for MISTRAL but much more provided the real-world problems which made the work interesting. Special thanks to Dr. Klaus Elhardt, Dr. Thomas Ruf, and Prof. Timos Sellis, Ph.D. for all the discussions that helped to combine theory and practice.

Thanks to Michael Bauer, Michael Muchitsch, and my colleagues, who did a great job in proofreading and in providing comments which helped to improve the presentation of the work.

Last but not least, I want to thank my family and my friends, especially my love Annette, for their support throughout the last years. They encouraged me in my pursuit, never complaining about the long days and many weekends in the office, but always making sure that I take my time to relax whenever it was necessary. Thanks for your patience.

Abstract

Multidimensional access methods are considered to be a promising approach for providing acceptable performance to analysis-centric applications. However, despite the large body of research work in this field, the commercial support for multidimensional indexes is still very weak. The reason for this discrepancy is threefold: first, no standard multidimensional index like the B-Tree for one-dimensional data has emerged so far. Second, integrating a new access method into a database system kernel is usually a complex and expensive task. Third, current query optimizers still have problems in dealing with multidimensional data making it difficult to use multidimensional indexes efficiently.

In this thesis, we address the above mentioned deficiencies and promote the *universal* B-Tree (UB-Tree) as a premier candidate for a general-purpose, multidimensional index.

In an extensive theoretical and experimental comparison with R*-Trees we show that the UB-Tree can compete with other approaches in multidimensional indexing. The UB-Tree outperforms the R*-Tree not only w.r. to query performance but also considering the important properties of maintenance performance, index size and others.

Addressing the kernel integration, we reveal another big advantage of UB-Trees: relying on the standard B-Tree as underlying structure the integration effort is reduced significantly. We further present optimizations of the basic algorithms, like the reduction of post-filtering, for range query processing, which have large impact in practice. To improve the support of query optimizers for multidimensional access methods, we propose a new type of multidimensional histograms based on UB-Tree concepts. At the same time, we point out general limitations of multidimensional synopses.

Finally, we introduce the concept of weighted dimensions, which allows for tuning of the UB-Tree to application specific preferences among the dimensions. This makes the UB-Tree viable to a broader range of applications. The concept also leads to an improved range query processing for standard composite key indexes.

Contents

List of Figures

List of Tables

Part I

Multidimensional access methods

Like a building, a thesis requires a solid foundation. In the first part of this work, we give an overview on the state-of-the art of multidimensional indexing and the relevant application domains. In addition, we introduce the basic terminology used throughout the thesis and we propose a comparison framework for index structures. Based on this framework, we present a comparison of R*-Trees and UB-Trees.

Chapter 1

Introduction

In the last two decades, emerging applications, like data warehousing, data mining, or geographic information systems, have brought up new requirements for database management systems (DBMSs). The basic query pattern has shifted from one-dimensional point queries (e.g., money transfer between two bank accounts) to multidimensional range queries (e.g., average sales of goods in Munich in December 2001). This change in the query paradigm is caused by the transition from transactional processing in *on-line transactional processing (OLTP)* systems to the more human-centered analytical processing common in the modern applications. Multidimensional range queries result from the various multidimensional modelling approaches, which are used in *on-line analytic processing (OLAP)* and many other application domains to allow for more intuitive formulation of queries by the user. The efficient support for such query patterns is a crucial aspect of today's DBMSs.

1.1 Multidimensional data is everywhere

Handling multidimensional data is an inherent problem of relational database management systems (RDBMSs). In the relational model [Cod70], each relation defines a multidimensional space given by the domains of the attributes. Each tuple represents a point in the multidimensional space. More general, each non-empty subset of the attributes of a relation defines a multidimensional space where the points have associated information represented by the other attributes. This analogy between relations and multidimensional space has long been neglected in literature.

Only with the move from transaction oriented applications, classified as OLTP applications, to more analysis centric applications the multidimensional model has attracted more attention. In OLAP applications the data model is typically multidimensional, representing the complex processes of the business world. In contrast to the simple bookkeeping operations in OLTP applications, OLAP applications have to support complex analysis queries including multiple attributes (features, parameters, dimensions, etc.) simultaneously. For example, for analyzing sales data, the date of the sale, the location of the sales (e.g., in which shops), and the type of product sold are of interest.

With the increasing importance of analytical queries, handling of multidimensional data becomes a crucial issue for DBMSs. Such queries commonly restrict multiple attributes of one relation and result in completely different access patterns as the typical point searches of OLTP applications. Consequently, index structures developed for OLTP applications do not provide the best performance for the new query types. New solutions are therefore necessary to cope with the new requirements.

Typical application domains

As mentioned before, due to the correspondence between relations and multidimensional space, all DBMS applications can be regarded to be multidimensional. A typical example for a simple scenario is the modelling of a $n-m$-relationship. Such relationships are represented by a separate table containing foreign keys to the two base tables. Joining the two base tables in combination with a restriction on both sides will lead to a multidimensional query on the $n-m$-relationship table. Subsequently, we want to focus on applications that are characterized by the multidimensional nature of their queries.

The first applications to be considered multidimensional, were applications handling spatial data. Ranging from geographic information systems (GIS) to modern location-based applications for mobile devices, such applications have to handle at least two-dimensional data. For example, in a mobile tourist information systems the user wants to get information about the attractions nearby the current location. The first developments in multidimensional indexing where motivated by this application domain.

Data warehouses (DW) are typical examples of OLAP applications: the so-called *fact* data is recorded in the context of multiple dimensions. Analytical queries usually restrict one or more of the specified dimensions, e.g., total sales of notebooks in a specific area, resulting in multidimensional restrictions on the fact data.

In data-mining, as an extension to data warehousing, a multidimensional modelling approach is also often applied in cluster analysis (in contrast to the logical approach based on association rules). In both application domains the cube analogy is often used at the user-interface level.

Handling XML data has been an increasingly important topic in recent years. Depending on the mapping of XML to relations the handling of queries on XML data also becomes a multidimensional problem requiring multidimensional indexing for efficient retrieval of documents.

1.1.1 State of the art

Despite the large research work on multidimensional access methods in the last years (see Chapter 3 for details) the support for such access methods in commercial DBMSs is still very poor.

B-Trees [BM72, Com79] are still the de-facto standard of index structures. Only a few systems have included multidimensional indexes, usually for handling spatial data. For example, Oracle and Informix Universal Server (now IBM) provide

strongly limited implementations of R-Trees in their extensions for spatial data.

An engineering approach to multidimensional indexing is to use several one-dimensional access methods and combine them in query processing. This so-called *index intersection* has the advantage that it is very flexible w.r. to the support of different attributes, but lacks the advantage of clustering the data w.r. to multiple dimensions.

The weak support of multidimensional access methods in DBMSs today results in our opinion from three problems: first, the kernel integration of new index structures is usually a very costly task, as the new functionality has to be coupled with essential database services like locking, logging, recovery, etc. Second, many proposed multidimensional indexes have been designed having special scenarios in mind, i.e., they do not support a wide range of applications. The database vendors, however, are looking for universal methods that can be applied to many applications. Third, current query optimization models do not explicitly carry over to multidimensional data and access methods. The main problem for today's cost-based optimizers is to get reliable estimates for cardinalities and cost of multidimensional operators/predicates. Even though various models/approaches exist for some multidimensional access methods, no general applicable, i.e., reliable, estimation method for multidimensional data exists.

1.1.2 The optimal multidimensional access method

In the previous section we have pointed out the deficiencies of state-of-the-art access methods. In order to evaluate our work, we have to set up the requirements for the optimal multidimensional index. The following list presents a rather informal description of important aspects of an access method. The requirements will be put in more concrete terms throughout this thesis.

- *Universality*:
  In order to allow for a wide range of applications the index structure has to be flexible w.r. to the support of various data types, e.g., the support for point objects as well as for extended objects (e.g., polygons, line segments, spheres, etc.).

- *Multi-user support*:
  For today's applications an efficient handling of concurrent operations is mandatory.

- *Symmetry w.r. to all dimensions*:
  All indexed dimensions should have the same weight/importance. That means, no dimension should be preferred in query processing unless explicitly wanted by the user. Consequently, access methods that allow for individual weighting of dimensions provide the highest flexibility.

- *Easy integration*:
  The cost-benefit ratio of the integration of new methods is an important factor

for commercial DBMS vendors. The easier a new index can be integrated into the existing system, the higher the probability of the integration. Both, the complexity of the new algorithms and the combination with the existing system play an important role for the ease of the integration.

- *Dynamism*:
  The dynamic nature of many applications also causes frequent changes to the underlying database. Therefore, the performance of a multidimensional access method should be independent of the update behavior of the application, i.e., the frequency, the number, and the order of insert, delete, and update operations.

- *Worst case guarantees*:
  With increasing data sizes, the predictability of response times becomes a significant issue. Access methods that provide worst case guarantees for each operation, allow for such predictions and make query optimization easier.

- *Low space complexity*:
  In times of tumbling hard disk prices, space requirements are no longer a limiting factor. Still, the less extra space an index structure requires the better, as the index size is an important influence factor for the performance.

1.2 Objective

As discussed above, despite their important role in providing sufficient performance for today's DBMS applications, multidimensional access methods are not widely available in commercial systems. The reasons for this fact are on the one hand the high cost of integrating index structures into a DBMS kernel and on the other hand the narrow usability of many multidimensional index structures. A standard multidimensional access method, like the B-Tree for one-dimensional data, is still missing.

This work demonstrates that the UB-Tree (universal B-Tree)[Bay97] overcomes the mentioned limitations of previous approaches. We show its high potential as a general-purpose multidimensional data structure as it combines flexibility, functionality, and efficiency with technology that is easy to integrate into existing systems. More precisely, we

- evaluate the UB-Tree w.r. to the requirements of a general purpose access method mentioned in the previous section,

- discuss all issues of integrating the UB-Tree into a DMBS kernel and provide experiences from the first integration into a commercial system, and

- present optimizations of the basic UB-Tree concepts and advanced concepts allowing for higher efficiency and usability.

We evaluate the UB-Tree based on a detailed comparison with R*-Trees. The comparison covers the relevant properties of an access method and thus allows for a reliable conclusion on the usability of the two prominent multidimensional access methods.

We cover in detail the kernel integration of UB-Trees, addressing the design and implementation of the basic algorithms as well as advanced topics like optimizer extension (focussing on the important issue of cardinality estimation).

Furthermore, we introduce optimizations and enhancements of the UB-Tree, which have great impact in practice. On the one hand, we optimize the range query performance by reducing post-filtering, achieving performance improvements for most applications. On the other hand, we introduce the concept of weighted dimensions, allowing for prioritizing some dimensions according to their importance. This also leads to improved processing of multidimensional range queries with standard B-Trees.

1.3 Structure of the thesis

What follows is a more detailed overview of the structure of the thesis combined with a few guidelines for the reader.

The thesis is divided into three parts: the first one introduces multidimensional access methods in general preparing the ground for the more detailed discussions in the later parts. The second part then covers the integration of the UB-Tree into a DBMS kernel and various optimizations. The last part introduces advanced multidimensional indexing concepts.

In Part I, after the general introduction, we define the basic terminology used throughout the thesis in Chapter 2. The reader familiar with the basic concepts in the indexing area finds a summary of the used terms and symbols in Section 2.6. We briefly discuss general related work in Chapter 3; specific related work to the discussed aspects is presented in the corresponding chapter notes. Chapter 4 introduces the running examples used for illustration throughout the thesis. The in-depth comparison of UB-Trees and R*-Trees in Chapter 5 concludes the first part.

Part II covers the integration and optimization of UB-Trees. Chapter 6 summarizes our experiences of integrating the UB-Tree into a DBMS kernel. Within this chapter some implementation details of the UB-Tree and its algorithms are described. Chapter 7 discusses the linearization of query boxes and its application for optimizing the range query algorithm. Further, we elaborate on the approximation of Z-regions by minimum bounding boxes in Chapter 8. The important issue of optimizer extension for UB-Trees covered in Chapter 9 completes the middle part.

Part III deals with advanced concepts of multidimensional indexing. We are presenting an enhancement of UB-Trees to allow for arbitrary weighting of indexed dimensions in Chapter 10. In Chapter 11 we show how composite key B-Trees can support multidimensional queries more efficiently by an improved range query algorithm. We summarize our contribution and point out directions of future work and at the end of this thesis.

Chapter 2

Foundations and terminology

In this chapter we introduce the basic terminology used throughout this thesis. We try to stay conform with [Mar99], however, in some cases we have to modify the definitions slightly for our purposes, but without changing the intuitive meaning.

We focus on indexing for relational database management systems (RDBMSs), even though our methods apply also to DBMSs in general. Therefore, we expect the reader to be familiar with the basic terminology of relational systems. In RDBMSs, data is organized in a set of *relations* or *tables*. Each relation consists of *tuples*, often referred to as *rows*, which are composed of multiple *attributes* or *columns*. A given relation has a fixed *arity*, i.e., a fixed number of attributes.

Our approach to multidimensional indexing considers tuples to be *points* in multidimensional space. The *coordinates* of the point are given by the attributes of the tuple. We refer to the attributes as *dimensions*. Often, we are only interested in a subset of the attributes of a relation R for our theoretical considerations. We then partition R into a set of *qualifying attributes* or *index attributes*, i.e., the attributes that specify the coordinates in the multidimensional space, and a set of *quantifying attributes* or *information attributes*, i.e., attributes that provide additional information. If nothing else is specified, we speak of qualifying attributes when we use the term attributes.

2.1 General notation

This section introduces the general notational conventions used in this thesis. If nothing else is specified, we use the standard terminology established in computer science.

Sets and Strings

Let $S = \{s_1, s_2, ..., s_n\}$ be an arbitrary set. $|S|$ denotes the *cardinality* or *size* of S. Analogously, for a string T, $|T|$ denotes the *length* of T. Given a string $T = t_1 t_2 \dots t_l$ of length l, $T_{1\dots m}$ denotes the prefix of length $m \leq l$ of T, i.e., $T_{1\dots m} = t_1 \dots t_m$.

Matrices

Let $\mathcal{M}$ be an (m,n)-matrix, i.e., $\mathcal{M} = \begin{pmatrix} m_{11} & \dots & m_{1n} \\ \vdots & \ddots & \vdots \\ m_{m1} & \dots & m_{mn} \end{pmatrix}$.
We sometimes write $\mathcal{M}_{ij}$ for element m_{ij} of $\mathcal{M}$.

Decimal Numbers

We use the European/German convention of representing floating point numbers: the comma ',' is the decimal delimiter whereas the period '.' is the thousands separator.

Bitstrings

Let $\mathbb{B}$ denote the domain of all bit strings, i.e., $\mathbb{B} = [0|1]^*$. For a bit string $b = b_{l-1}b_{l-2}\dots b_0 \in \mathbb{B}$, of length $|b| = l$, b_{l-1} denotes the most significant bit. More precisely, the bit $b_i, 0 \leq i \leq |b|-1$ has the decimal value 2^i. Consequently, the whole bit string b has the decimal value $\sum_{i=0}^{|b|-1} b_i * 2^i$. We use $b_{\dots k}$ to denote the prefix of bit string b up to the bit position k, i.e., $b_{\dots k} = b_{l-1}b_{l-2}\dots b_k$; consequently, $b_{\dots k}$ has length $|b_{\dots k}| = l-k$. $b_{k\dots m}$ denotes the substring of b from position k to m, i.e., $b_{k\dots m} = b_k b_{k-1}\dots b_m$.

2.2 Multidimensional domain, relation, tuple

Let $\mathbb{D}$ be a **domain**, i.e., a finite set of values, with a total ordering $<_{\mathbb{D}}$. $min^{\mathbb{D}}$ and $max^{\mathbb{D}}$ denote the minimal and the maximal value of the domain. If the meaning is clear from the context, we write $<$ instead of $<_{\mathbb{D}}$. For each domain $\mathbb{D}$ there exists an order-preserving bit representation, i.e., each element a of the domain can be expressed by a bit string $a_{l-1}\dots a_0$ of length l[1] and the lexicographic order on the bit strings adheres to the natural order $<_{\mathbb{D}}$ of the domain.

Definition 2.1 (<-neighbor)
For any domain $\mathbb{D}$ with ordering relation $<_{\mathbb{D}}$ two values $a, b \in \mathbb{D}$ are ***<-neighbors****, iff $a < b \wedge \nexists c \in \mathbb{D}$ with $a < c < b$.*

Definition 2.2 (Order-preserving bit representation)
For each domain $\mathbb{D}$ a mapping β to bit strings can be defined: $\beta : \mathbb{D} \rightarrow \mathbb{B}$. We call $\beta(a)$ the ***bit representation*** *of $a \in \mathbb{D}$. The bit representation β is* ***order preserving*** *iff*

$$\forall a_1, a_2 \in \mathbb{D} : a_1 \leq_{\mathbb{D}} a_2 \Leftrightarrow \beta(a_1) \leq_{\mathbb{B}} \beta(a_2)$$

Let R be a *relation* with n attributes A_1, ..., A_n of domains $\mathbb{D}_1$, ..., $\mathbb{D}_n$. Without loss of generality, let A_1, ..., A_d $(d \leq n)$ be the index attributes of R and $A_{d+1}, ..., A_n$ be the information attributes of R. R is a set of tuples $t = (t_1, ..., t_n)$, where $t_i \in \mathbb{D}_i$;

[1] Leading 0 bits are used to fill up all bitstrings of the elements of one domain to the same length.

$|R|$ is the *cardinality* of R. We write $t.A_i$ synonymously for t_i for a given tuple t. R^I denotes the projection of R w.r. to the index attributes, i.e., $R^I = \{(t_1, ..., t_d) | \exists s \in R : s_1 = t_1 \wedge ... \wedge s_d = t_d\}$.

Definition 2.3 (Multidimensional domain, dimensionality)
*The **multidimensional domain** Ω of a relation R is the cross product of the d domains of the indexing (qualifying) attributes, i.e., $\Omega = \mathbb{D}_1 \times \ldots \times \mathbb{D}_d$. We say Ω and R are of **dimensionality** d.*

We call Ω also the **base space** or the **universe** of R, as R^I is a subset of Ω, i.e., $R^I \subseteq \Omega$. The cardinality or volume of Ω is the product of the cardinalities of the domains.

Definition 2.4 (Volume of Ω)
*The **volume** $vol(\Omega)$ of the multidimensional domain Ω is given by the product of the sizes of the d domains, i.e., $vol(\Omega) = \prod_{i=1}^{d} |\mathbb{D}_i|$.*

Sparsity of a data set

For the analysis of data sets, we are often interested in how dense the universe is populated for a given relation. We therefore have to distinguish between the *cardinality* and the *volume* of a relation.

Definition 2.5 (Volume of a relation)
*The **volume** $vol(R)$ of relation R with universe Ω, is the volume of Ω, i.e., $vol(R) = vol(\Omega)$.*

With this definition we now can define the *sparsity* of a relation that describes how densely populated the multidimensional universe is.

Definition 2.6 (Sparsity of a relation)
*Given a relation R of the multidimensional domain Ω, the **sparsity** ξ of R is $\xi(R) = 1 - \frac{|R|}{vol(\Omega)}$. $\frac{|R|}{vol(\Omega)}$ is called the **density** of R.*

Example 2.1: Volume, Density, and Sparsity

Let R be a three-dimensional relation with a base domain $\mathbb{D}$ (i.e., $\Omega = \mathbb{D} \times \mathbb{D} \times \mathbb{D}$) with $|\mathbb{D}| = 2^3$. Consequently, $vol(\Omega) = vol(R) = 2^3 * 2^3 * 2^3 = 2^9 = 512$. The following table shows the density and sparsity of R depending on the number of tuples in R. It is important to note that in real world applications data sets are usually very sparse ($\xi \gg 99\%$; see Chapter 4 for real world examples).

◇

Table 2.1: Density and sparsity depending on the size of R

$\|R\|$	**Density**	**Sparsity**
0	0	1
128	0,25	0,75
256	0,5	0,5
384	0,75	0,25
512	1	0

Order of the multidimensional domain

It is sometimes necessary to have an order in the multidimensional space. In the following we define a partial order on the points in multidimensional space based on the total order in the individual domains.

Definition 2.7 ($\trianglelefteq$-order of Ω)
For the multidimensional domain Ω we define the partial order $\trianglelefteq$ as follows:

$$\forall x, y \in \Omega : x \trianglelefteq y \Leftrightarrow \forall i, 1 \leq i \leq d : x_i \leq y_i$$

$$\forall x, y \in \Omega : x \triangleleft y \Leftrightarrow \forall i, 1 \leq i \leq d : x_i < y_i$$

Definition 2.8 ($\triangleleft$-neighbors)
*Two points $x, y \in \Omega$ with $x \triangleleft y$ are **$\triangleleft$-neighbors** iff $\exists i, 1 \leq i \leq d$ such that $\forall j, 1 \leq j \leq d, j \neq i : x_j = y_j$ and x_i, y_i are $<$-neighbors.*

2.3 Queries, result sets, and selectivity

In the following, we define the terminology dealing with queries on a relation. We limit our examination to restriction queries, i.e., queries that only select a subset of one table without considering projections.

Definition 2.9 (Query, result set)
*A restriction **query** is a predicate $\rho(t)$ on the tuples t of a relation R. All tuples satisfying ρ are called the **result set**, i.e., $RS(\rho, R) = \{t \in R | \rho(t)\}$.*

The query predicate ρ defines a subspace $Q \subseteq \Omega$ of the universe, with $Q = \{t \in \Omega | \rho(t)\}$. Thus, the result set of a query can be also specified as $RS(\rho, R) = Q \cap R$. For notational convenience, we often write $Q(R)$ instead of $RS(\rho, R)$ and speak of query Q on relation R, wherever the meaning is clear from the context.

Definition 2.10 (Selectivity of a query)
*The **selectivity** sel of a query Q on a relation R is defined as the fraction of the result set size over the size of R, i.e., $sel(Q, R) = \frac{|Q(R)|}{|R|}$.*

The term selectivity often causes confusion when it is used informally: a very selective query, i.e., a query with a small result set, has a small selectivity according

to our definition. We will therefore often speak about *small*, i.e., very selective, and *large* queries, i.e., only somewhat selective. In order to distinguish the two aspects of a query, we also define the term *volume of a query*.

Definition 2.11 (Volume of a query)
*The **volume** $vol(Q)$ of a query Q is the volume of the multidimensional space defined by the query.*

For dense or uniformly distributed data sets, there is a strong correlation between the volume and the selectivity of a query. For sparse data sets the relationship of the two aspects strongly depends on the data distribution and with that on the location of the query box.

In this work we concentrate on a special type of queries, namely *multidimensional range queries*. Multidimensional range queries are characterized by an interval restriction on each attribute. Let R be a relation with d attributes A_1, ..., A_d with corresponding domains $\mathbb{D}_1, ..., \mathbb{D}_d$, respectively.

Definition 2.12 (Multidimensional range query)
A multidimensional range query RQ on relation R is specified by the predicate $\rho(t) = l_1 \leq t.A_1 \leq h_1 \wedge l_2 \leq t.A_2 \leq h_2 \wedge ... \wedge l_d \leq t.A_d \leq h_d$. In general, ρ is specified by a multidimensional interval $[[l, h]] = [l_1, h_1] \times ... \times [l_d, h_d]$ and we write $RQ = [[l, h]]$.

We often refer to RQ as query box, because $[[l, h]]$ specifies an iso-oriented query window or box on Ω. The volume of a multidimensional range query $RQ = [[l, h]]$ is the given by $vol(RQ) = \prod_{i=1}^{d}(h_i - l_i + 1)$. $h_i - l_i + 1$ denotes the number of values in the interval $[h_i, l_i]$; the length can be determined for each interval of a domain $\mathbb{D}$ as we are only considering finite domains with a fixed ordering.

Multidimensional range queries are of great importance as they can be used to specify various interesting query types:

- *Exact match queries* or *point queries* PQ restrict all dimensions to a point, i.e., $PQ = [[x, x]]$

- *Partial match queries* PM restrict some dimensions to a point while the other dimensions are not restricted, i.e., $PM = [[l, h]]$ where $l_i = h_i$ or $l_i = min^{\mathbb{D}_i}$ and $h_i = max^{\mathbb{D}_i}$

More complex query shapes are usually handled by a range query corresponding to the minimal bounding box (MBB) of the shape combined with appropriate post-filtering with the exact predicate given by the shape. Another way of specifying such complex queries is to use a union of range queries, i.e., using a decomposition of the complex query shape.

2.4 Physical organization of data: clustering and access methods

For the discussion of access methods it is also necessary to consider the physical organization of relations on secondary storage. Figure 2.1 shows the typical architecture of a hard disk drive. One drive consists of a *stack* of spinning disks, each of which consists of groups of *blocks* forming one *track*. A block is the smallest addressable unit of transfer for secondary storage, with block sizes varying between 512 bytes and a few KBs.

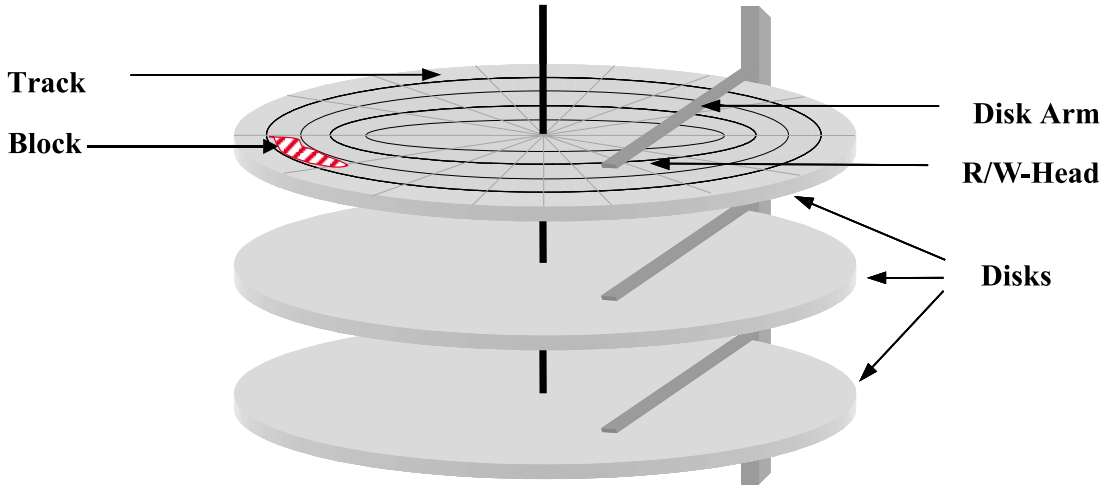


Figure 2.1: Physical structure of a hard disk

To read one block, the *read/write-head (R/W-head)* has to be positioned over the track of the block and reads it when it passes underneath the R/W-head.

We distinguish two different accesses types which result in different access times:

1. *random access*
2. *sequential access*

Each block access consists of two components: on the one hand the I/O part for fetching the block from secondary storage and on the other hand a CPU part for post-processing the fetched block. Correspondingly, the total access time for a block t is subdivided into the time $t_{I/O}$ for I/O and the time t_{CPU} for CPU processing. In the following, we concentrate on the I/O time, as it is still the dominating factor in today's computer systems.

For a random access, the R/W-head is positioned to the correct track resulting in *positioning time* t_π. This time covers the track positioning and the latency until the block is rotated underneath the R/W-head. Once the R/W-head is positioned, it takes t_τ to transfer the block into main memory. Hence, $t^{rand}{=}t_\pi + t_\tau$. Once the R/W-head is positioned on one track, the subsequent blocks can be read sequentially only causing $t^{seq} = t_\tau$. The typical specifications of today's hard drives [IBM01] with data rates between 36,6 MB/sec and 52,8 MB/sec, resulting in $t_\tau \approx \frac{1}{50}$ ms,

and average seek times t_π between 3,4 ms and 8 ms identify the positioning of the R/W-head as the dominating part of the access time.

To balance the vast difference between t_π and t_τ, today's operating and database systems map a constant number of subsequent blocks to a (*database*) *page*. Page sizes between 2KB and 256KB are typical in modern systems.

As tuples are usually smaller than a page, multiple tuples are stored on one page by the DBMS, i.e., a page can be regarded as a container for tuples.

Definition 2.13 (Page, page content, and page utilization)
*A **page** P is a container for a set (or multiset) T of tuples, the **content** of P. P has a fixed **capacity** C^P, i.e., $0 \leq |T| \leq C^P$. The **page utilization** U of page P is defined as $U = \frac{|T|}{C^P}$.*

Whenever the meaning is clear from the context, we use P to denote both aspects of a page: the content of the page or the container itself.

2.4.1 Clustering

As for most applications the I/O cost is still the limiting factor, the organization of the data on secondary storage has great influence on the overall processing time. Clustering is an important concept to significantly reduce the I/O component by reducing the number of disk accesses. The goal of clustering is to store data that is likely to be processed together physically close together on secondary storage, i.e., on the same page, if possible and thereby to reduce the number of I/O necessary to retrieve the wanted data.

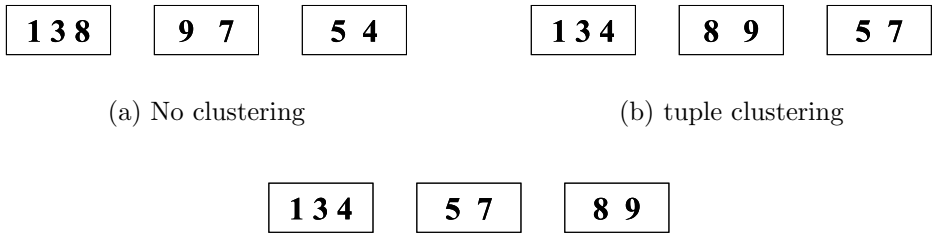
The following definition is taken from [Mar99].

Definition 2.14 (Tuple clustering, page clustering)
***Tuple clustering** stores tuples of one or several relations on one page, if the tuples are likely to be used together to create the result set of a query. If the tuples do not fit on one page, the tuples have to be stored on several pages. Normally new pages are physically placed on secondary storage in insertion order. **Page clustering** in addition to tuple clustering also maintains physical clustering between pages.*

Example 2.2: Clustering

To illustrate the concept of clustering we use the domain of integers $\mathbb{N}_0$. The standard ordering $<$ will define the 'closeness' criteria for the clustering. Figure 2.2 shows the three clustering types for the integer set {1,3,4,5,7,8,9} stored on three pages.

Now, consider a query that asks for all values between 3 and 7, i.e., the range $[3, 7]$. If the data is not clustered (Figure 2.2(a)) all three pages have to be accessed. With clustering, however, only 2 pages have to fetched from disk (Figure 2.2(b), Figure 2.2(c)). The advantage of page clustering (Figure 2.2(c)) is that the two pages are stored consecutively on disk such that sequential access is possible, i.e., only one expensive positioning of the R/W-head is required.



(a) No clustering

(b) tuple clustering

(c) page clustering

Figure 2.2: Clustering of integers

◇

2.4.2 Access methods

An *access method* or *index* is a way to organize data on secondary storage. For each relation R or, more general, for each set of data there exists one clustering access method that defines the physical organization on the storage device. In addition, multiple non-clustering access methods can be defined on R. Depending on the index, different ways of accessing the data are available. Unfortunately, there exists a somewhat confusing terminology about index structures in the database community. We clarify the meaning of terms used throughout this work below.

According to [GR93], we can distinguish between non-associative and associative addressed file organization. The simplest non-associative data organization is the *sequential file*, where the tuples are stored in insertion order or entry sequenced. This organization only allows for unqualified access to the data, as the entire file has to be read to answer a query. This is called *full table scan (FTS)* or short *scan*. More sophisticated access is provided for associative data structures, where the data is organized according to the *key* or *address* of the tuples. The term key has two interpretations in the database world: first, it refers to the *logical key* of a relation, i.e., the set of attributes that uniquely identifies a tuple. Secondly, it is used in the meaning of a *search key* for data structures. Throughout this thesis, key is always used in the meaning of a search key.

Definition 2.15 (Address of a tuple)
Let $\mathbb{A}$ be the domain of addresses. The address function $\alpha : \Omega \rightarrow \mathbb{A}$ maps each tuple t of a relation R with base space Ω to an ***address*** *$\alpha(t)$.*

Often, we just write α in short for $\alpha(t)$. For example, B-Trees use one attribute of the tuple or a concatenation of multiple attribute values as the key of the tuples; R*-Trees use the minimum bounding boxes (cf. Section 5.2.1) as address of the tuples. The ambiguous meaning of the term key often leads to confusing naming practice: a *primary index* often refers to a clustering index with the logical key of the relation as address. *Secondary indexes* usually are non-clustering indexes. If the key of the index is a concatenation of several attributes of the relation, one

speaks of a *compound* or *composite* key. In this thesis, we just distinguish between *clustering* and *non-clustering* indexes. While a clustering access method specifies the organization of the data on secondary storage, i.e., direct access to the data is provided, non-clustering indexes only allow for indirect access via tuples references.

Current DBMSs support three major classes of access methods:

- *Hash indexes*: the data is partitioned into an array of buckets and a hash function on the key of tuples is used to identify the corresponding bucket. Hash indexes efficiently provide direct access to tuples, but do not support range queries.

- *Tree-based indexes*: B-Trees [BM72] and its variants are by far the most used index structures in DBMSs. R-Trees are used in some systems for the organization of multidimensional, extended objects. The TransBase RDBMS [TAS01] supports UB-Trees for handling multidimensional point data.

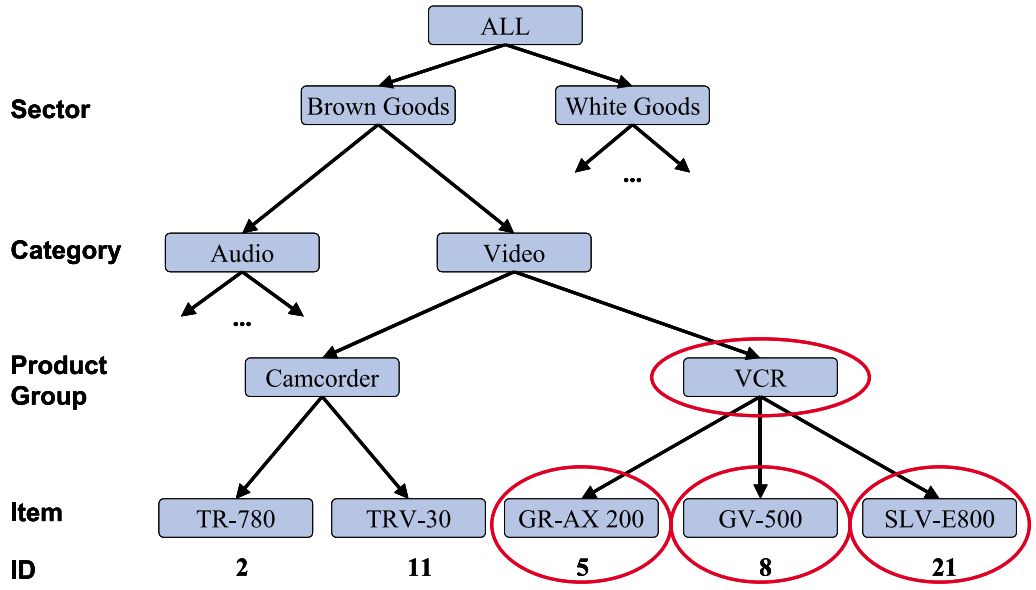
- *Bitmap indexes*: non-clustering method using a special representation of the tuple references, allowing for efficient evaluation of multi-attribute restrictions [OG95, OQ97].

In Chapter 3 we discuss multidimensional index structures in more detail.

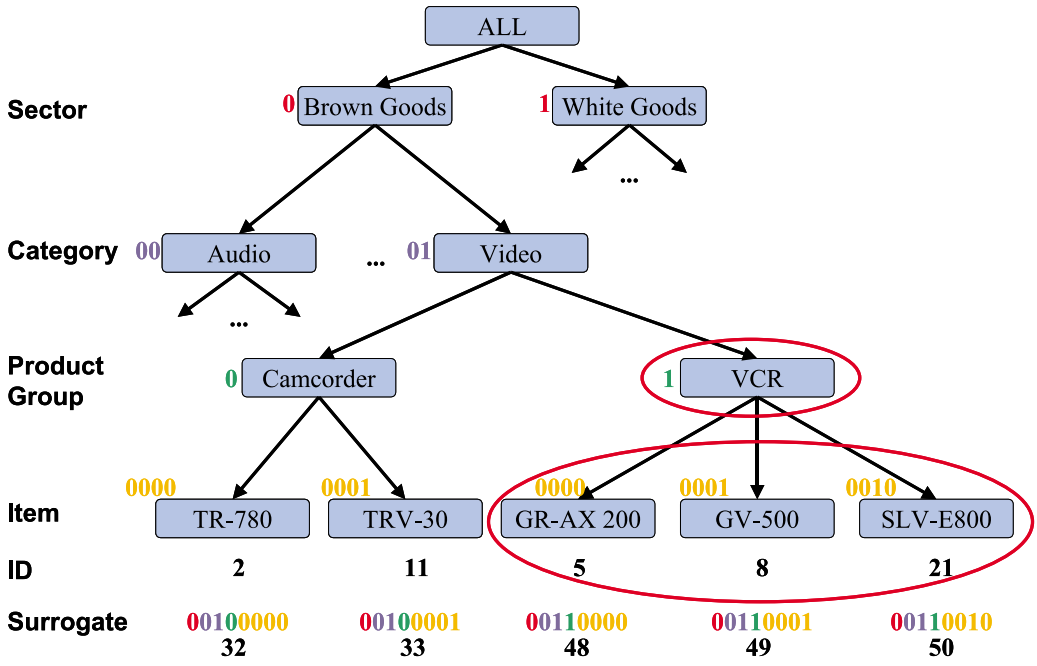
2.5 Clustering of hierarchies

Hierarchies play an important role in various application domains. They are used to provide a semantic structure to data, e.g., a geographical classification of customers in a data warehouse. As the hierarchies cover the application semantics they are used frequently by users to specify the restrictions on the data as well as the level of aggregation. The restrictions on the hierarchies usually result in point or range restrictions[2] on some hierarchy levels [Sar97]. The problem that arises is that these restrictions on upper hierarchy levels lead to a large set of point restrictions on the lowest level, i.e., the level with the most detailed information. This situation is depicted in Figure 2.3(a): restricting the level 'Product Group' to the value 'VCR' leads to the set of ids $\{5, 8, 21\}$ and not to the interval $[5, 21]$, as the item with id 11 does not belong to the specified product group. For most access methods it would be more efficient to process one range restriction instead of a set of point restrictions. The resulting question is how to map a point/range restriction on a higher hierarchy level to a range restriction on the lowest level? To this end, we have proposed a clustering scheme for hierarchies in [MRB99]. New keys for the elements on the lowest level are generated which reflect the hierarchy semantics, i.e., keys which adhere to the partial order defined by the hierarchy levels. These so-called *(compound) surrogates* guarantee that the keys of all elements in a sub-tree of the hierarchy lie within a closed interval (cf. Figure 2.3(b)) such that a key of an element

[2] Range restrictions on hierarchy levels are only meaningful if an order on the elements of the hierarchy level is defined.



(a) Non-clustered hierarchy



(b) Clustered hierarchy

Figure 2.3: Example of hierarchy clustering

not lying in the subtree is not within the interval. In our example, the restriction to the product group 'VCR' now leads to the interval $[48, 50]$; the item with id 11 is mapped to the surrogate 33 that does not violate the interval. We refer to this technique as **hierarchy clustering (HC)** from now on.

2.6 Terminology in a nutshell

In the following table we list all important terms and symbols defined in the previous section and which are used throughout the thesis with this meaning.

Table 2.2: Terms and symbols

Concept	**Symbol**
One-dimensional domain	$\mathbb{D}$
minimal/maximal value of a domain	$min^{\mathbb{D}}/min^{\mathbb{D}}$
Dimensionality	d
Multidimensional domain	Ω
Multidimensional range query	$Q = [[min, max]]$
Result set of query Q on relation R	$Q(R)$
Selectivity of a query Q on relation R	$sel(Q, R)$
Volume of a space S	$vol(S)$
Sparsity	ξ
Length of (bit)string	l
Cardinality of a set S	$\|S\|$
Address of a tuple t	$\alpha(t)$
Page capacity	C
Page utilization	U

Chapter 3

Related work

This chapter provides a brief overview of work related to this thesis. We have decided to discuss specific work related to the issues covered in this dissertation in extra sections for each chapter. We hope to facilitate the reading as related work is discussed directly in the context of our approaches.

In general, the related work covers the whole area of query processing and multidimensional access methods.

3.1 Query processing in relational DBMSs

The access to the base tables is a crucial aspect in query processing, because it influences the overall processing strategy. A concise survey of query processing techniques can be found in [Gra93]; [Cha98] provides an overview of query optimization.

One fundamental rule in query processing says that restrictions should be performed as soon as possible, in best case already on the base tables. How efficient such restrictions can be evaluated depends on the organization of data on secondary storage and on the functionality of the access method.

Without indexing, all data has to be scanned in order to evaluate a restriction. While this way of access requires high overhead for small queries, it is still used frequently in modern systems as the combination of page clustering and increasing disk performance compensates much of the theoretical drawback. A widely used rule of thumb states that a full table scan performs better as soon as more than 10% of the data has to be retrieved.

The B-Tree [BM72] and its variants (B^+-Tree, B*-Tree; cf. [Knu98], [Com79]) are by far the most used tree-based index structures in database systems. We will use the term B-Tree to refer to the B^+-Tree within this thesis. Clustering B-Trees store the tuples directly in the leaf nodes whereas non-clustering B-Trees only store references to the tuples. B-Trees allow for point and range queries and guarantee a worst-case storage utilization of 50%. For efficient support of multi-attribute restrictions, multiple single-attribute, non-clustering B-Trees are commonly used. In this approach, which is often named *inverted file*, retrieval time is favored over maintenance time and storage complexity. Hash indexes [FNPS79, Lit80] are a good alternative for point queries, but they do not support range query access.

Bitmap indexes [OG95, OQ97, CI98] are a non-clustering access method using a special representation of the tuple values and references. Single bitmaps store the information whether an attribute has a specific value or not. This scheme allows for efficient evaluation of multi-attribute restrictions just using bit operations but lacks the advantage of clustering. For range queries on bitmap indexes various advanced encodings have been proposed [WB98].

With the numerous ways of indexing a table, finding the best organization becomes a further optimization issue. The problem of *index selection* is an important task of *physical database design*, which in addition covers topics like partitioning or replication of data. Index selection is often discussed for special application domains, e.g., OLAP [GHRU97, Sar97], and more generally in tools of commercial DBMSs [CN97, CN98a, CN98b].

3.2 Multidimensional access methods

As mentioned in the introduction, there exists a large body of work in the area of multidimensional access methods. [GG98] provides an excellent overview of the field. Data structures are classified according to the supported data types into *point access methods* (PAMs) and *spatial access methods* (SAMs). In the following we assume the organization of data volumes that do not fit into main memory. Therefore, we restrict our discussion to secondary storage structures. Main memory data structures require different design as I/O is no longer the limiting factor [LC86a, LC86b, BMK99]. With increasing buffer sizes in today's systems, the work on cache-conscious index implementation has recently attracted new attention [GL01, KCK01, BMR01].

3.2.1 Point access methods

The characteristics of point data, i.e., having no extension in the multidimensional space, allows for special optimized indexing compared to the handling of extended objects.

PAMs can be categorized into three classes with the following prominent examples (for details we refer to [GG98]).

- *Hashing:* grid files [NHS84], EXCELL [Tam82], two-level grid file [Hin85], twin grid file [HSW88], and multidimensional hashing [Fal86, Fal88]

- *Tree-based structures:* K-D-B-Tree [Rob81], LSD-Tree [HSW89], Buddy Tree [SK90], BANG File [Fre87], hB-Tree [LS90]

- *One-dimensional mappings:* combination of space filling curves and one-dimensional index structures. [Sag94] provides a survey on space filling curves; the zkd-B-Tree [OM84] and the linear clustering method of [Jag90] are examples.

3.2.2 Spatial access methods

[RSV01] includes a very recent overview on SAMs. The majority of access methods can further be categorized according to the internal organization into *space-driven* and *data-driven* structures. Space-driven structures use a fixed partitioning of the complete space to organize the tuples, whereas data-driven structures adapt the partitioning according to the real data distribution. A further classification of SAMs is done according to the way extended objects are handled [GG98].

- *Transformation*: One way of handling extended objects is to transform them into a one-dimensional representation and use any PAM for indexing. Besides the objects the queries have to be transformed as well.
  - *Dual Space*: extended objects are mapped to points in higher-dimensional space, the dual space or parameter space [Ore90]. There are different mappings, e.g., endpoint, midpoint, that lead to different query shapes and data distributions in dual space [FMB00, FSR87].
  - *Decomposition*: Instead of mapping extended objects to higher-dimensional points, space-filling curves are used to approximate the object. Generally, the object is described by a set of one-dimensional intervals [OM84, Ore89a, Ore89b, Gae95]. This approach overcomes the problem of high dimensionality of the previous method, but introduces redundancy.
- *Overlapping Regions*: the concept of overlapping regions allows for non-redundant storage of extended objects. Usually, the complex object is approximated by a simpler shape, e.g., a minimum bounding box. The R-Tree [Gut84] and the R*-Tree [BKSS90] are the most prominent representatives of this approach. Data structures using multiple layers are a special variant of overlapping regions trying to reduce the effects of large objects (e.g., multi-layer grid file [SW88]).
- *Clipping*: Overlapping regions lead to problems of unpredictable query performance, because multiple paths in the index may have to be traversed. To prevent this, clipping-based schemes partition the objects storing them in disjoint buckets at the cost of introducing redundancy (e.g., R^+-Tree [SRF87], CELL-Tree [Gün89]).

3.2.3 Data structures for high-dimensional spaces

An even more complex task is the indexing for high-dimensional data. [WSB98] has introduced the infamous *curse of dimensionality* that states that clustering becomes meaningless for high-dimensional spaces due to the lack of useful distance metrics. This observation has led to new approaches for high-dimensional spaces, mostly trying to combine dimensionality reduction schemes with traditional multidimensional indexing [CM00, CM99a, BBK98].

3.3 Previous work in the MISTRAL project

This dissertation is based on the work done in the MISTRAL (**M**ultidimensional **I**ndexes for **St**orage and for the **R**elational **Al**gebra) project at the Bavarian Research Center for Knowledge-Based Systems (FORWISS) [MIS].

The cornerstone of the MISTRAL project is the publication of the basic concepts of the *universal* B-Tree (UB-Tree) [Bay96, Bay97]. Based on this work, the UB-Tree has been intensively studied and enhanced throughout the MISTRAL project. [Mar99] provides an in-depth theoretical analysis of the UB-Tree as well as a first practical evaluation based on a prototype implementation. [MZB99] introduces the Tetris-algorithm for processing of queries with sort operations and [FMB99] extends the important range query algorithm to non-rectangular query shapes. [RMF+00] describes the integration of the UB-Tree into the kernel of a commercial DBMS, which was the objective of the MDA project funded by the European Commission. [RMF+01] presents the results of applying the UB-Tree to the data warehouse application of GfK (cf. Section 4.3). In the follow-up project EDITH [EDI], the integration of hierarchical clustering into a DBMS kernel as well as the development of special query processing algorithms for such cases is currently carried out.

Chapter 4

Running examples

Throughout this thesis we will use a real-world data warehouse application to illustrate the theoretical concepts. The data warehouse application will also provide the framework for performance measurements and comparisons. Even though we concentrate on data warehouse applications in this thesis, the results also apply to other application domains like geographic information systems, archiving systems, and much more. To support this, we use additional real-world data sets for some of our analyses. Table 4.1 lists the data sets used throughout this work.

Table 4.1: Real world data sets

Name	**Dimensionality**	**Comment**
GEO	2	2-dimensional, GIS data
CENSUS5D	5	real-word data from the US census bureau
GFK3D	3	real-world market research data warehouse; HC encoded; see Section 4.3 for details
GFK11D	11	real-world market research data warehouse; see Section 4.3 for details

As Table 4.1 shows, we use data sets of different dimensionality to also cover the influence of the dimensionality on the index structure. In the following sections we briefly introduce the used applications and data sets.

4.1 The GEO data set

The geographical data set (GEO) consists of GPS coordinates of interesting points in the city of Lodz in Poland. It is a subset of a local utility (i.e., water, heating, electricity etc.) provider data warehouse. The distribution of the 2-dimensional point data is depicted in Figure 4.1.

GEO consists of $\approx$ 370.000 tuples (á 100 bytes); the sparsity of GEO is very high: $\xi(\text{GEO}) > 99,9999\%$.

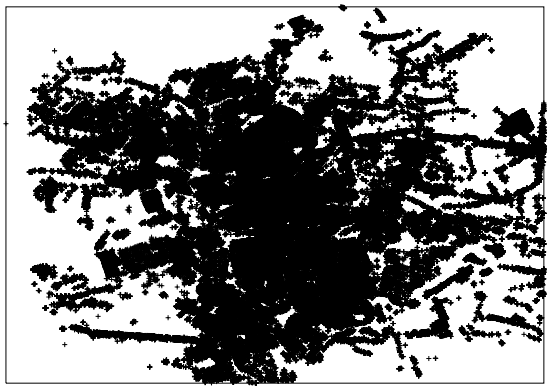


Figure 4.1: Data distribution of the GEO data set

4.2 The census data set

The census data from the US Census Bureau [USC] is taken from the Current Population Survey (CPS) data source. From this survey we have taken the Person Data Files of the March Questionnaire Supplement, which are also used in other publications (e.g., [DGR01]). From the data of the years 1994 - 2000 we extract the 5-dimensional data set CENSUS5D. Table 4.2 shows all attributes and the size of the corresponding domains of the selected data set.

Table 4.2: Census data attributes

Name	**Domain size**	**Comment**
SEX	2	sex of person
CTZSHP	5	citizenship of person
EDU	17	educational attainment
HOURS	88	hours usually worked at main job
AGE	91	age of person

CENSUS5D contains 59K tuples (á 100 bytes). Due to the smaller universe compared to GEO, the sparsity is somewhat smaller: $\xi(\text{CENSUS5D}) = 99,64\%$.

4.3 The GfK non-food panel data warehouse

The "Gesellschaft für Konsumforschung" (GfK) is the largest market research institute in Germany and among the top ten world wide. As running example we use a subset of the non-food panel data warehouse. This GfK data warehouse tracks the sales of non-food goods (e.g., refrigerators, TVs, etc.) according to three dimensions: Time, Product, and Segment. The Segment dimension describes the point of sales, i.e., the shops from which the sales data is collected. The goal of the warehouse is to provide in-depth analysis (like market share, best-sellers, trends etc.) of the market.

The products are hierarchically classified according to sectors, categories, and product groups. The lowest granularity in the time dimension for this example is

the two-month period (time required to get a representative sample of the market), classified according to four-month periods and years. For the Segment dimension multiple hierarchies (e.g., turn-over classes, organizational classifications) exist, but the most important one is the geographical classification of the shops according to countries, regions, and micro markets. Figure 4.2 shows the abstract star schema of the GfK data warehouse (GfKDW).

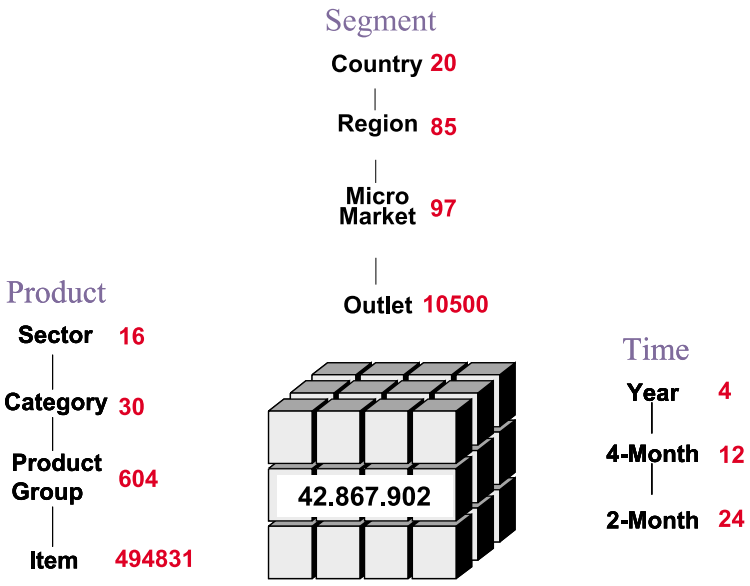


Figure 4.2: GfK DW Star schema

The used snapshot of GfKDW stores around 43 million fact records (approx. 4 GB) associated with 24 two-month periods, 10.500 shops, and more than 490.000 products.

This abstract star schema can be modelled differently for the relational implementation: on the one hand we can use a combination of all 11 hierarchy fields to index the fact table, allowing for restrictions on arbitrary hierarchy levels. On the other hand, we can use the hierarchy clustering approach HC to generate semantic keys for each dimension, also allowing for arbitrary hierarchy restrictions. Both resulting universes are very large, leading to very sparse data set: $\xi(\text{GfKDW}) \gg 99,9999\%$.

4.3.1 Flat-schema: GfK11D

If we want to support arbitrary hierarchy restrictions directly on the fact table, we have to include all hierarchical attributes in the fact table. This corresponds to a completely denormalized modelling of the star schema (actually it is no longer a star then).

4.3.2 HC-schema: GfK3D

We exploit the semantics of the hierarchies to improve query performance by applying hierarchical clustering (HC) (see Section 2.5) to the GfK data warehouse. With HC, point restrictions to any hierarchy level always lead to more preferable range restrictions on the lowest level, i.e., on the fact table. All access methods on the fact table benefit from HC as a range query is more efficient to handle than a set of point queries. The hierarchies of the GfKDW are encoded with HC (cf. Section 2.5) leading to following domain sizes: $|\text{Product}| = 2^{29}$, $|\text{Segment}| = 2^{24}$, and $|\text{Time}| = 2^5$. This results in a very large universe of size 2^{58}. A detailed data distribution analysis for this data set is provided in Appendix A.

4.3.3 Queries on GfKDW

For GfKDW, like for most OLAP applications, the dimension hierarchies provide the key navigation paths for interactive OLAP, allowing for meaningful query formulation via drill-down, roll-up, or slice-and-dice operations [CD97]. For example, a typical query to determine the best selling laptop in the last period in Germany would restrict the Time dimension on the two-month level, the Segment dimension on the country level, and the Product dimension on the product group level.

The GfK has defined a set of reports and ad-hoc queries on its DW. Each report falls into one of three groups: hitlists, running reports, segmentation reports (or feature splits). Hitlists provide several measures for the items within one product group or category, and sort the items by one of the reported measures. Hitlists are a ranking of items with respect to one measure in a single period (cf. Figure 4.3).

Retail Audit - Germany - Segmentation hitlist

Camcorder; April/May Panelmarket

	SUM(Sales)	MIN(Price)	MAX(Price)	AVG(Price)	SUM(Turnover)
Item 50027	**232**	**749**	**999**	**930,28**	**216056,96**
Item 50035	**171**	**639**	**849**	**827,23**	**141456,33**
Item 40011	**144**	**1179**	**499**	**1368,65**	**197055,60**
...	**...**	**...**	**...**	**...**	**...**

Figure 4.3: Segmentation hitlist example

Running reports differ from hitlists in that, instead of different measures in the columns of the report, the columns show the same measure in different periods. Their rows can be grouped according to product features. Segmentation reports, like hitlists, show different measures for one period, but their rows are groupings according to features like in running reports. An example segmentation report is shown in Figure 4.4.

Retail Audit - Germany - Camcorder; April/May					Segmentation report Panelmarket
	SUM(Sales)	MIN(Price)	MAX(Price)	AVG(Price)	SUM(Turnover)
Total	**247218**	**499**	**3699**	**837,18**	**206965965,24**
Hi8 mono	**18527**	**499**	**729**	**638,77**	**13687191,79**
Hi8 stereo	**107936**	**749**	**3699**	**1691,92**	**182619077,12**
...	**...**	**...**	**...**	**...**	**...**

Figure 4.4: Segmentation report example

The three report types so far make up 90% of the analyses delivered by GfK to their clients. Besides these fixed reports, GfK is moving more and more to ad-hoc analysis. Ad-hoc analysis differs from 'static' reports in the way that usually a set of subsequent drill operations is executed, which have the same context. For example, a user starts a session with asking the total sales for a specific segment in a given country and in a given two month period. A drill-down to a region or a product category provides more detailed information. Finally, the user wants to compare the numbers with the previous period. As consequence, ad-hoc analysis usually generates drilling patterns where the restrictions on one dimension change (e.g., going down the hierarchy or switching to the sibling) while the restrictions on the other dimensions do not change. However, as user behavior is not predictable, sometimes so-called *random* queries are placed, which are used to navigate to a completely different 'location' in the cube [Sap01].

Chapter 5

Where is the general-purpose, multidimensional index?

After covering all preliminaries in the previous chapters, we now turn to the first objective of this thesis: the question about the general-purpose, multidimensional access method. The research community has answered the increasing need for multidimensional indexing with a huge variety of new data structures (cf. Chapter 3). Among those data structures the R-Tree family is by far the most prominent one, but no real standard, like the B-Tree for one-dimensional data, has been established so far. This stems from the fact that many data structures have been designed having special applications in mind, not fulfilling all requirements we set up in the introduction. Besides finding out which data structures come close to a universal multidimensional index, the question arises, which one of the large set of indexes to choose for a certain purpose.

Comparing different data structures has always been a major task on the way of developing new indexing methods. However, most of these comparisons have not followed a certain methodology for comparing data structures, but only have concentrated on query performance figures on different data distributions. For an efficient usage in real-world systems, however, much more criteria than just the query performance of an index are of importance. For instance, multi-user support is essential and in many systems, even in data warehouses, insertion or load performance cannot be neglected. Furthermore, the comparison should not only be based on experimental results, but should also contain a theoretical analysis to support the experimental findings. As consequence, a comprehensive and comprehensible comparison of data structures is necessary to assess the overall performance of index structures.

This chapter presents a comparison of R*-Trees and UB-Trees using a comparison framework for index structures that fulfills the above mentioned requirements. We include various criteria, like query performance, maintenance performance, and index size, into our model and consider objective metrics for the I/O performance as well as for the CPU performance.

The contribution is two-fold: first, we present a framework for comparing different index structures according to all aspects relevant in practice using objective metrics. Second, we present a thorough comparison of two multidimensional access

methods, namely the UB-Tree and the R*-Tree. In this comparison we do not only include query performance, but according to our comparison framework also maintenance performance, and index size. The comparison results show that the UB-Tree outperforms the R*-Tree with respect to index size, query and maintenance performance. Our results are not only based on experiments, we support them by a theoretical analysis of both index structures.

The chapter is organized as follows. After presenting the comparison framework in Section 5.1, Section 5.2 introduces the basic concepts of UB-Trees and R*-Trees. In Section 5.3 we present the results of the theoretical and experimental evaluation.

5.1 A comparison framework for multidimensional access methods

In order to get an objective comparison of index structures, we present a framework taking all relevant properties into account. Based on the results w.r. to the individual characteristics, one is then able to generate an overall rating, weighting the properties according to the needs of the application.

5.1.1 Comparison criteria

Even though query performance is undoubtedly the dominating criteria, other properties of index structures are also important for real-world applications (e.g., insertion performance). These properties have to be considered if one wants to assess the overall quality of an index structure. For this reason we include the following properties into our comparison framework:

- index size,
- maintenance performance,
- query performance,
- dimensionality,
- flexibility regarding supported data types, and
- multi-user support

The index size, maintenance performance, and query performance are data dependent properties and we will discuss them in more detail in the following subsections. Dimensionality, multi-user support, and flexibility, however, are more qualitative properties of an index structure. Even though most multidimensional access methods have no limit on the dimensionality in theory, there are usually limitations in practice due to the 'curse of dimensionality'. Some data structures have been designed only for multidimensional point data whereas others are especially developed for handling extended objects. Multi-user support is extremely important in real-world applications. As consequence, only index structures with appropriate locking

and logging mechanisms can be considered to have practical relevance. Efficient multi-user support is often much more important than query performance. We will now discuss in more detail the rest of the proposed criteria.

5.1.1.1 Size of the index

In times of decreasing hard disk prices, space requirements of an index are often considered not to be an important factor any more. However, if we are thinking about databases on tertiary storage, e.g., a product catalog distributed on CD-ROM or DVD-ROM, space consumption is again an important factor. In addition, the index size directly influences the performance of all operations on the index (e.g., an index with twice the amount of occupied disk space will take twice the I/O to return all data). We will distinguish between the size of the *index part* and the size of the *data part* of the index structure. On the one hand, the size of the data part is of major importance as the more disk pages are required to store the data the more disk I/O is usually necessary for retrieving the data. On the other hand, the size of the index part influences to which extent the index part can be cached by the DBMS. The lesser index nodes are cached the more disk I/O on the index is required.

5.1.1.2 Maintenance performance

In most application scenarios, even in many data warehouses, data is not static. As consequence, the maintenance performance, i.e., the time for insertion, deletion, and update, should also be considered for the overall performance of the index. One has to distinguish two types of maintenance patterns: *random* and *bulk* operations. Random operations typically result from interactive user requests and therefore have to be as efficient as queries. Bulk operations are usually triggered by batch jobs, which are expected to take longer. As the maintenance windows are getting smaller and smaller due to the increasing requirements on the availability of the systems, efficient processing of batch operations is also needed. Bulk operations do not only comprise insertion (e.g., periodical loading of new data into a DW), but also deletion (e.g., deletion of old periods after archiving). Updates of the index attributes are regarded as a combination of deletion and insertion. Updates of non-indexed, fixed-length attributes are comparable with queries, with the only difference that the accessed pages also have to be written back to secondary storage.

5.1.1.3 Query performance

As mentioned previously, query performance is the key aspect of an index structure. We consider two types of queries: multidimensional range queries and multidimensional point queries (as a special case of the former). The query performance of a multidimensional access method is basically influenced by two factors:

- the index size and
- the clustering technique of the index.

With a good clustering strategy, I/O is reduced significantly as data that is likely to be accessed together is placed close together on secondary storage. The goal is to find a clustering method that works fine for a given query profile and that is robust against different data distributions. If no knowledge is known about the query profile, i.e., all queries have the same probability, a symmetrical data structure is more flexible with respect to arbitrary restrictions in multiple dimensions. The chosen clustering scheme usually has direct impact on the index size. Up to now, multidimensional range queries define the most important query pattern on multidimensional data. However, with new query types arising from novel applications, e.g., nearest neighbor queries [RKV95] or skyline queries [BKS01], it becomes interesting to also compare the support of data structures for such operations, in the future.

5.1.2 Comparison metrics

After defining the properties, we specify how we are going to compare the access methods according to them.

The index size is best measured in number of pages S occupied by the index. We will refer to the number of index pages as S_{Index} and to the number of data pages as S_{Data}, i.e., $S = S_{Index} + S_{Data}$. Further, we include the average page utilization U (see Definition 2.13) as a metric. U_{Index} denotes the average page utilization of the index nodes, while U_{Data} denotes the average page utilization of the data nodes.

Two factors influence the maintenance and query performance: *I/O cost* and *CPU cost*. With the increased main-memory buffers of today's systems, the CPU cost of database operations has been identified to be increasingly important for the overall system performance [BMK99, GL01], so that it should be included in the comparison. Obviously, execution times are no appropriate metric for comparing the performance, as too many unknown factors (e.g., the quality of implementation) influence this metric. Consequently, we use the number of page accesses P as the metric for the I/O cost and the number of key comparisons K for the CPU cost of an operation. In case of tree-based data structures, K basically is proportional to the number of nodes that have to be evaluated during an operation[1]. As different index structures use different representations of keys it is necessary to normalize the cost for a key comparison (e.g., on average one R*-Tree key comparison is more than twice as expensive as a key comparison for UB-Trees; cf. Section 5.4).

Table 5.1 lists the defined criteria and corresponding metrics of our framework.

5.2 Basic concepts of R*-Trees and UB-Trees

To answer the question about the universal multidimensional index, one would need to compare all proposed access methods, but this is an almost impossible task.

[1]For non-tree indexes, like bitmap indexes, K should reflect the CPU cost of the dominating operations, like the intersection of bitmaps.

Table 5.1: Overview of the used metrics

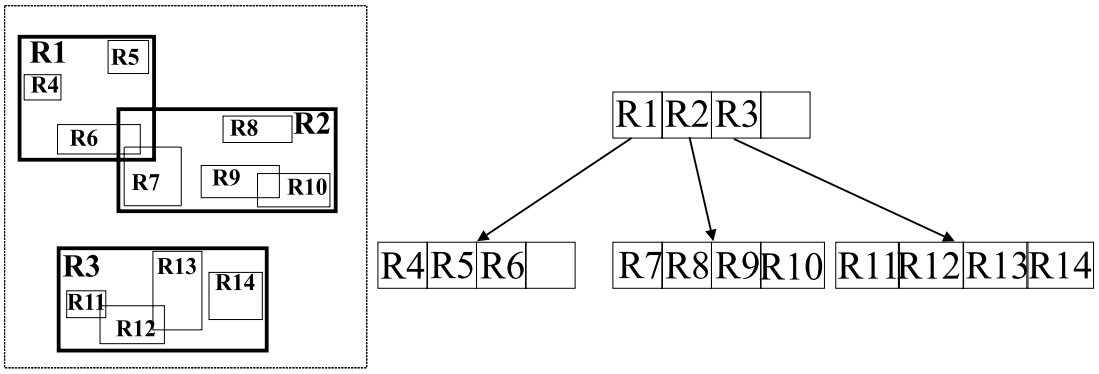
Criteria		**Metric**
Index size	index part	S_{Index}
	data part	S_{Data}
	height	h
Page Utilization	index part	U_{Index}
	data part	U_{Data}
Performance	I/O cost	P
	CPU cost	K

Therefore, we have chosen to restrict our comparison of the UB-Tree to the toughest competitor, the R*-Tree. The R*-Tree (or the R-Tree family in general) is widely accepted in academia and industry. It is not restricted to specific application domains and provides good performance for low dimensionality.

In the following, we briefly introduce the basic concepts of the two index structures concentrating on the properties we use for our theoretical and experimental analysis.

5.2.1 Basic concepts of R*-Trees

The R*-Tree [BKSS90], like the standard R-Tree [Gut84], uses the concept of overlapping regions to store objects in a single bucket/region. The objects are described by their d-dimensional *minimum bounding box* (MBB), often also called *bounding rectangle*. With this concept, the R*-Tree can handle point data as well as extended objects. An R*-Tree corresponds to a hierarchy of nested MBBs, each MBB corresponding to a node of the tree. The MBBs of all descendants of a parent node are contained in the corresponding MBB of the parent. MBBs on the same level of the tree are allowed to overlap. Like for the B-Tree, each node of the R*-Tree is mapped to one disk page and the R*-Tree is height balanced, i.e., each path from the root to a leaf has the same length. The standard R-Tree has problems to adapt to various data distributions. As consequence, the R*-Tree provides an improved insertion algorithm to overcome this deficit. The concept of *forced reinsertion* is used to achieve a better space partitioning. Instead of splitting a node as soon as it overflows, a certain number of tuples is removed from the node and is re-inserted into the tree. The forced reinsertion can be regarded as a kind of periodic reorganization of the tree. In addition, the splitting algorithm has been improved with respect to minimizing the overlap between MBBs and improving the space utilization of the nodes. However, reducing the overlap and increasing the node utilization are contrary goals. As consequence, a lower average page utilization as for B-Trees is achieved. [BKSS90] suggest 30% for the reinsertion factor and a minimal page utilization guarantee of 40%. Figure 5.1 shows a space partitioning for an R*-Tree and the corresponding tree.



(a) Space partitioning of a 2-dimensional R*-Tree

(b) R*-Tree structure

Figure 5.1: Bounding boxes and corresponding R*-Tree

Optimization for point data

The R*-Tree can be easily optimized for handling only multidimensional point data like we plan to do for our comparisons. Instead of storing a complete MBB for a data point one just stores the point on the data pages, eliminating the overhead of MBBs completely. On the index level, however, one still requires MBBs.

Basic operations

The basic operations of the R*-Tree follow a similar principle as for B-Trees: for each node it is tested, which path to follow according to which MBBs are intersected by the query MBB. Due to the concept of overlapping MBBs, however, multiple paths of the R*-Tree may have to be traversed in order to find the correct leaf node. This significantly influences the query performance (in worst case all nodes of the tree may have to be evaluated). In best case, however, only the root node has to be evaluated. This happens when the query falls into a part of the universe that is not populated (often referred to as *dead space*) and therefore may not be covered by one of the top level MBBs. The range query processing is similar to the point query method: starting from the root, all nodes are traversed whose MBBs are intersected by the query rectangle.

Bulk loading

As noted above, the insertion phase is critical for the R*-Tree performance. The insertion algorithms directly influence the page utilization as well as the overlap between MBBs. For huge data sets, the random insertion algorithm of R*-Trees is too expensive and does not result in optimal trees. As consequence, different methods for bulk-loading/packing R-Trees have been proposed. The two most prominent packing algorithms are the Hilbert curve based algorithm from [KF93] and the STR approach from [LEL97]. Both methods create the R-Tree in a bottom-up manner and require the input data to be sorted either according to the Hilbert-value or ac-

cording to one coordinate of the rectangles. For all these approaches, the clustering strategy is given by the order of the data generated by the algorithms. However, this clustering is not maintained by the R*-Tree, and the question arises how the quality of the clustering behaves after successive random inserts on a packed/bulk-loaded R*-Tree. More recent approaches use buffering techniques for bulk operations on R*-Trees [dBSW97, AHVV99].

5.2.2 Basic concepts of UB-Trees

We just give a short introduction to UB-Trees here, details are presented in Chapter 6. The basic idea of the UB-Tree [Bay97] is to use a space-filling curve to map a multidimensional universe to one-dimensional space. Using the Z-curve for preserving multidimensional clustering it is a variant of the zkd-B-Tree [OM84].

A Z-address $\alpha = Z(x)$ is the ordinal number of the key attributes of a tuple x on the Z-curve, which can be efficiently computed by bit-interleaving. A standard B-Tree is used to index the tuples taking the Z-addresses of the tuples as keys. The fundamental innovation of UB-Trees is the concept of Z-regions to create a disjunctive partitioning of the multidimensional space. This allows for very efficient processing of multidimensional range queries. A Z-region $[\alpha, \beta]$ is the space covered by an interval on the Z-curve and is defined by two Z-addresses α and β. Each Z-region maps exactly onto one page on secondary storage, i.e., to one leaf page of the B-Tree.

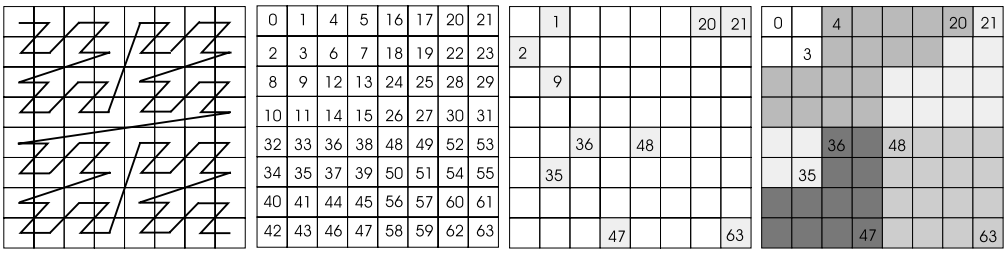


Figure 5.2: Z-curve and Z-regions

Figure 5.2(b) shows the ordinal numbers of the two-dimensional tuples on the Z-curve (Figure 5.2(a)). Figure 5.2(d) shows a Z-region partitioning created by the point distribution (points are represented by their Z-addresses) shown in Figure 5.2(c) assuming a page capacity of 2 points: we get the 5 Z-regions [0,3], [4,20], [21,35], [36,47], and [48,63].

Basic operations

The processing of basic operations, i.e., insertion, deletion, update, and point query, of the UB-Tree are analogous to the basic operations of the B-Tree. For each tuple the corresponding Z-address is computed, and with the resulting value

the underlying B-Tree is accessed. Thus, all basic operations require only cost proportional to the height of the tree, i.e., $O(\log S_{Data})$. The only modification to the standard B-Tree algorithms necessary is the adaptation of the splitting algorithm to achieve a good, i.e., as rectangular as possible, Z-region partitioning.

Range query processing

Based on the Z-region partitioning of the data, the range query algorithm computes the minimal Z-region cover of the query box, i.e., the set of all Z-regions that properly intersect the query box. The algorithm does not require any I/O of data pages for its computation and has a linear complexity with respect to the length of the Z-address (see Chapter 6 for details).

Bulk loading

Bulk loading for UB-Trees is similar as for B-Trees. The data to be inserted is sorted according the the Z-address of the tuples. Then, the pages are filled up-to the desired space utilization and written to disk. [FKM+00] proposes an improved method for the case where data is to be bulk inserted into an already filled UB-Tree.

5.3 Comparison environment and general results

In the following sections we present our theoretical analysis and will support our findings with experimental results. We will concentrate on the comparison of the index size, the maintenance, and the query performance. The obtained results also give hints regarding the robustness of UB-Trees and R*-Trees with respect to increasing dimensionality. However, we restrict our analysis to up-to 11 dimensions as [WSB98] shows that in high dimensional cases a simple sequential scan on average outperforms any clustering multidimensional access method. We will briefly address the other comparison criteria below.

5.3.1 Supported data

Both data structures support any data type with a total ordering. While the UB-Tree only handles multidimensional point data, the concept of MBBs allows R*-Trees for also handling extended objects. However, in combination with the *dual-space* approach, it is possible to handle extended objects represented by MBBs efficiently with the UB-Tree as well (cf. [FMB00]).

5.3.2 Multi-user support

As UB-Trees rely on the proven technique of B-Trees, efficient algorithms for locking and logging already exist. Recently, various concurrency control algorithms have been proposed for R-Trees [NK94, KB95, CM99b] or for the more general GIST framework [KMH97]. Even though these concepts also apply to R*-Trees, the concept of reinsertion severely reduces the degree of concurrency that can be achieved.

We consider the number of data pages that have to be locked for an insertion to be the critical factor for the degree of concurrency - the more pages have to be locked the lesser the degree of concurrency. For the UB-Tree only one data page has to be locked in any case. The same holds in the best case for the R*-Tree, but the number of locked pages increases in case of reinsertion. In worst case, all re-inserted tuples lead to new page accesses and consequently to additional required page locks. For example, in the GfK3D application, with a page capacity of 25 tuples and a reinsertion degree of 30%, this leads to an lock overhead of up to 8 pages per insertion. Our results show (cf. Section 5.5) that on average the insertion of one tuple requires more than 4 data page I/O for R*-Trees with reinsertion.

5.3.3 Comparison environment

For our comparisons we use the real-world data sets described in the previous chapter. With respect to the indexes, we use the following implementations of the two data structures:

- UB-Tree (UB): integrated in the TransBase DBMS [TAS01]
- R*-Tree (RST): optimized for multidimensional point data (based on the original implementation used in [BKSS90])

In order to investigate the various tuning possibilities for R*-Trees, we use different configurations (see Table 5.2) in our experiments. We are varying the minimum guaranteed page utilization and the reinsertion percentage, resulting in the following naming scheme for R*-Tree instances: RST_<min page utilization>_<reinsertion percentage>.

Table 5.2: R*-Tree configurations

Name	**Description**
RST_40_30	optimal R*-Tree configuration according to [BKSS90]
RST_50_0	corresponds to the standard settings of a UB-Tree
RST_50_30	same page utilization guarantees as a UB-Tree but using improved insertion strategy, i.e., reinsertion degree of 30%

5.4 Index size analysis

As we have mentioned before, the index size is a crucial factor for the performance of an index structure. The larger the index, the higher the cost of index operations. The more data pages, the more page accesses are required during query processing.

We usually restrict our comparison of query performance to data page accesses as it is usually safe to assume that the relevant index part is cached during query processing. However, if the index part gets larger and less index pages can be cached, the overhead for accessing index pages increases, as well.

In the following we do not consider compression, neither of data pages nor of index pages. To the best of our knowledge, none of the two data structures gains significant advantage w.r. to index size from one of the various compression techniques proposed by other researchers (e.g., [BU77, GRS98]).

5.4.1 Size of the data part

Both data structures store only the tuples and no additional information on the data pages. The UB-Tree omits storing the Z-values along with the tuples on the data pages, as they can be efficiently computed from the tuples via bit-interleaving (cf. Section 6.1.2). Likewise, the point-data optimized R*-Tree does not need to store MBBs for points. Consequently, one expects no differences in the number of data pages S_{Data}.

As our experimental results show in Section 5.4.3, this expectation is valid for bulk loaded R*-Trees and UB-Trees. But after random-insertion, especially for non-uniform data, the R*-Tree achieves a significantly lower page utilization compared to the UB-Tree. The resulting R*-Trees require more space and more page accesses in query processing, as we will see later.

5.4.2 Size of the index part

The size of the index part depends on three factors: the fan-out of the index part, the average page utilization of the index, and the number of data pages. Let B denote the size of an index page in bytes and A_{UB}, A_{RST} the address/key size in bytes of the UB-Tree and the R*-Tree, respectively. The size of an index entry E is, assuming a four byte reference to the child node, $E = A_{\{UB|RST\}} + 4$. Thus, the maximum fan-out F of the index part is $F = \lfloor \frac{B}{E} \rfloor$. The difference in the fan-out for UB-Trees and R*-Trees is a result of different key sizes. The R*-Tree stores complete bounding boxes as keys, i.e., two values per dimension. As consequence R*-Tree keys require two times the space of the Z-values indexed by the UB-Tree:

$$A_{RST} = 2 * A_{UB} \Rightarrow E_{RST} = (2 * A_{UB} + 4) = \frac{2 * A_{UB} + 4}{A_{UB} + 4} * E_{UB} \approx 2 * E_{UB}$$

For instance, assuming 4 bytes for each dimension, $E_{RST} = 1,67 * E_{UB}$ for 2 dimensions and $E_{RST} = 1,90 * E_{UB}$ for 10 dimensions. Hence, the fan-out F_{UB} of UB-Trees is roughly twice the fan-out F_{RST} of R*-Trees: $F_{UB} \approx 2 * F_{RST}$.

The difference in fan-out has immediate influence on the height of the index and thus on the number of index nodes. Let S_{Data} be the number of data pages; the index height h can be calculated from the fan-out F and the average page utilization

U_{Index} of the index nodes:

$$h = \lceil \log_{F*U_{Index}}(S_{Data}) \rceil$$

Given the index height h, the number of index nodes S_{Index} is computed as:

$$S_{Index} = \sum_{i=1}^{h-1} \left\lceil \frac{S_{Data}}{(F * U_{Index})^i} \right\rceil$$

Example 5.1: Index size simulation

Assuming the same number of data pages S_{Data} and the same average page utilization U_{Index} for UB-Trees and R*-Trees, which is actually a best case scenario for R*-Trees (cf. Section 5.4.3), Table 5.3 shows the difference in index height and the number of index nodes for a six dimensional data set (tuple size is 100B) with B=2KB, E_{UB}=28B, and U_{Index}=67%. The R*-Tree requires more than twice as many index pages and is often one level higher than the UB-Tree.

Table 5.3: Simulation of index sizes

# Tuples	h_{UB}	S^{UB}_{Index}	h_{RST}	S^{RST}_{Index}
10.000	2	15	3	34
100.000	3	139	3	314
500.000	3	687	4	1565
1000.000	3	1372	4	3127
5.000.000	4	6856	4	15627
10.000.000	4	13709	5	31253
50.000.000	4	68533	5	156253
100.000.000	5	137065	5	312502

◇

The theoretical analysis of the size of the index part already gives an indication that the R*-Tree will exhibit worse query performance than the UB-Tree due to the following two reasons:

1. Less caching efficiency: as the index part is larger, lesser index nodes can be cached leading to more I/O during query processing.

2. Higher CPU overhead: the higher the tree the more index nodes have to be evaluated during query processing. This directly leads to higher CPU cost.

5.4.3 Experimental results

For the experimental evaluation, we use bulk loaded instances of our test data sets. To capture the influence of different insertion techniques we also consider data sets created by random insertion into the mass-loaded instances. Table 5.4 presents the results of the experiments.

Table 5.4: Index sizes for the different data sets

Data set	Number of tuples	Type of insertion	Index	h	S_{Index}	S_{Data}	U_{Index}	U_{Data}
GEO	300000	BL	UB	4	184	20000	90%	80%
			RST	4	286	21430	72%	74%
GEO	371436	BL+ 70K	UB	4	276	23997	80%	83%
			RST_40_30	4	328	25159	72%	78%
			RST_50_0	4	368	25817	65%	76%
			RST_50_30	4	331	25122	72%	78%
CENSUS5D	50000	BL	UB	3	23	2381	96%	81%
			RST	4	72	2501	77%	77%
CENSUS5D	59341	BL + 9K	UB	3	29	2823	90%	81%
			RST_40_30	4	121	3349	65%	68%
			RST_50_0	4	106	3216	65%	71%
			RST_50_30	4	106	3214	65%	71%
GfK3D	13771716	BL	UB	4	6643	598771	92%	80%
			RST	5	10068	573823	79%	80%
GfK3D	19771716	BL+ 6M	UB	5	9021	813056	83%	85%
			RST_40_30	5	27531	1089998	60%	59%
			RST_50_0	5	21885	997781	66%	66%
			RST_50_30	5	21842	997440	66%	66%
GfK11D	13771716	BL	UB	5	15101	860733	95%	81%
			RST	6	53798	860734	78%	81%
GfK11D	19771716	BL+6M	UB	5	21807	1213968	91%	81%
			RST_40_30	7	132987	1559350	53%	63%
			RST_50_0	7	111544	1447600	63%	68%
			RST_50_30	7	111808	1450853	60%	68%

Our experimental results strongly support our theoretical analysis. For packed, bulk-loaded indexes both index structures show the same size. The R*-Tree sometimes even requires less data pages despite the same page utilization, as the UB-Tree implementation in the DBMS kernel requires more overhead (larger page header) which leads to a lower page capacity. The index part of the R*-Tree, however, is significantly larger (more than a factor of 3). The difference in the index page utilization, which stems from the fact that we can not control the page utilization of the index pages in the case of bulk loading for the integrated UB-Tree, is only a minor factor. The differences are the effect of the larger index entries for the R*-Tree.

As soon as additional data is randomly inserted, the R*-Tree is not capable of maintaining the packing achieved by the bulk loading. The index page and the data page utilization are declining. In contrast, the UB-Tree is growing much slower, keeping a page utilization of around 80%. Especially the difference in the index size is growing. Requiring more than 5 times more index pages than the UB-Tree, the R*-Tree will benefit much lesser from caching. Also, the resulting R*-Trees are in two cases higher than the UB-Trees; for GfK11D there is even a height difference of two!

5.5 Insertion performance analysis

To assess the insertion performance of the two data structures, we analyze the following insertion scenarios:

- creation of the index by random insertion or random insertion into a bulk loaded index
- creation of the index by bulk loading

5.5.1 Random insertion

For random insertion, the performance differences between R*-Trees and UB-Trees stem from the reinsertions performed by the R*-Tree. Re-insertions cause additional CPU and I/O cost for each insertion.

5.5.1.1 CPU Performance

Inserting a tuple into a tree of height h requires $h-1$ index node accesses. For each of these accesses the keys of the node have to be compared against the search key in order to identify the paths, which have to be traversed down further. As the UB-Tree stores the keys in Z-order, binary search can be applied, and therefore the number of key comparisons K is $\log_2 F_{UB}$ per node, i.e., $K_{UB} = (h_{UB}-1) * \log_2 F_{UB}$ for an insertion that does not cause a page split. For the R*-Tree, however, each key entry of an index node has to be evaluated, as no order on the keys is defined, resulting in $K_{RST} = (h_{RST}-1) * F_{RST}$. Even though $F_{RST} = \frac{F_{UB}}{2}$, the R*-Tree requires over 10 times more key comparisons per accessed index node as the UB-Tree for realistic fan-outs.

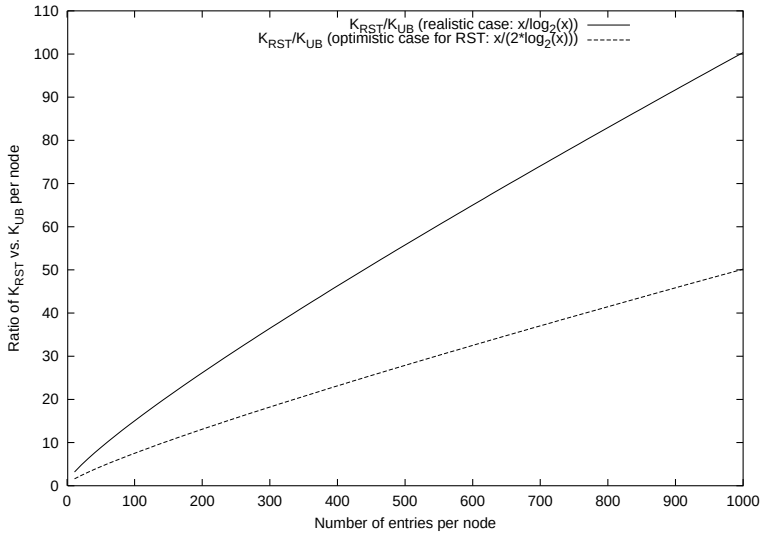


Figure 5.3: Ratio of key comparisons per node

Figure 5.3 shows the ratio of the key comparisons of a UB-Tree and an R*-Tree for one index node depending on the number of index entries on the node. The

best case for the R*-Tree (dashed graph) reflects the fact that the R*-Tree has half the fan-out of the UB-Tree (assuming same page utilization as for the UB-Tree). However, to get a realistic comparison, we have to consider that a key comparison for the R*-Tree is on average more than twice as expensive as a comparison for the UB-Tree (the intersection of two MBBs has to be tested). Assuming that an R*-Tree comparison corresponds to two UB-Tree comparisons, leads to the realistic ratio for the key comparisons depicted by the solid graph. It is important to mention that this CPU cost analysis applies for all search operations on the index.

In case of page overflows, we have to consider the overhead of page splitting, which may propagate up to the root. Assuming that copying key entries comes with the same cost as key comparisons, the worst case insertion performance (an actual growing of the tree) for UB-Trees is:

$$K_{UB} = \underbrace{(h_{UB}-1) * \log_2 F_{UB}}_{insertion} + \underbrace{(h_{UB}-1) * \frac{F_{UB}}{2}}_{\text{split overhead: copying } \frac{1}{2} \text{ of the tuples}}$$

At the same time, R*-Trees apply reinsertion to improve the organization of the tree, i.e., to reduce the overlap between the MBBs. Accordingly, the R*-Tree with reinsertion factor r requires

$$K_{RST} = \left(\underbrace{1}_{insertion} + \underbrace{r * F_{RST}}_{reinsertion} \right) * (h_{RST}-1) * F_{RST}$$

in best case, i.e., a page split is prevented due to reinsertion. If n page splits caused by the reinsertion propagate up the tree one requires

$$K_{RST} = (1 + r * F_{RST}) * (h_{RST}-1) * F_{RST} + \left((h_{RST}-1) * \frac{F_{RST}}{2} \right) * n.$$

The actual worst case for the R*-Tree occurs if all reinsertions cause an overflow of the corresponding data pages.

5.5.1.2 I/O Performance

For the I/O analysis of insertion we concentrate on the number of data page accesses[2]. Without reinsertion, a single insertion generally causes two I/O operations (one read + one write) in best case, and one additional I/O in case of an overflow. The total number of I/O operations required to insert n tuples for the UB-Tree is therefore $2 * n + \Delta S_{Data}$, where ΔS_{Data} is the number of pages created by the insertion. An R*-Tree has to expand $r * F_{RST} * U_{Index}^{RST}$ paths in case of reinsertion with a reinsertion factor of r, leading to additional I/O. Even without reinsertion,

[2]With today's main memory the complete index part, i.e., everything besides the leaf nodes, is usually cached.

the R*-Tree requires more I/O due to overlapping regions. Table 5.5 shows the reinsertion overhead for the data sets. The number of performed (perf.) I/O operations is included as well as the number of required (req.) I/O operations assuming no reinsertion and no overlapping regions.

Table 5.5: Overhead of Re-Insertion per insertion

Data	**Index**	**perf. I/O**	**req. I/O**	**Overhead Factor**
GEO	RST_40_30	6,15	2,07	2,96
[average over	RST_50_0	4,97	2,08	2,38
70K insertions]	RST_50_30	6,24	2,07	3,01
CENSUS5D	RST_40_30	4,92	2,09	2,35
[average over	RST_50_0	3,61	2,08	1,74
9K insertions]	RST_50_30	4,65	2,08	2,24
GfK3D 14M	RST_40_30	5,32	2,08	2,55
[average over	RST_50_0	3,57	2,07	1,73
1M insertions]	RST_50_30	4,70	2,07	2,27
GfK3D 18M	RST_40_30	5,97	2,08	2,87
[average over	RST_50_0	7,66	2,07	3,71
1M insertions]	RST_50_30	7,12	2,07	3,45
GfK11D 14M	RST_40_30	15,34	2,12	7,25
[average over	RST_50_0	7,62	2,10	3,63
1M insertions]	RST_50_30	9,95	2,10	4,74
GfK11D 18M	RST_40_30	62,75	2,12	29,66
[average over	RST_50_0	11,52	2,10	5,49
1M insertions]	RST_50_30	10,36	2,10	4,94

The amortized cost analysis results in more than 4 page accesses per inserted tuple for R*-Trees with reinsertion, while without reinsertion and without overlapping regions the amortized cost of random insertion is very close to the optimum of 2 I/O operations per tuple. It is interesting to note, that the random insertion performance deteriorates further, as more tuples are inserted, i.e., the larger the R*-Tree gets. For example, inserting into the GFK11D data set with 18 million tuples takes more than 5 times longer for the R*-Tree than for the UB-Tree (or any other data structure without reinsertion), independent of the R*-Tree configuration.

Summarizing our results, we have to state that random insertion is a severe performance bottleneck of R*-Trees, caused by the concepts of overlapping regions and reinsertion. This makes R*-Trees almost unsuitable for 'dynamic' applications.

5.5.2 Bulk loading

In order to handle the large volumes of data typical for today's applications, methods for efficient bulk operations are essential for any multidimensional access method. Especially the initial creation of an index is important, but also bulk updates and deletions are becoming more and more important. If one wants to compare different bulk loading algorithms one has to focus on two issues: the cost for the creation and the quality of the resulting index. With respect to the creation cost it has been shown (e.g., in [AHVV99]) that creating good trees requires at least the same cost

as external sorting. The algorithms for R*-Trees as well as for UB-Trees mentioned in Section 5.2.1 and Section 5.2.2 are optimal in this sense with one exception: the STR algorithm recursively sorts the data set d times causing higher costs.

The UB-Tree bulk loading algorithms naturally achieve the same clustering as repeated insertions but can guarantee the target page utilization. The known bulk loading algorithms for R*-Trees mainly differ in the quality of the created index, depending on the clustering technique used in the algorithm. In order to investigate the influence of bulk insertion on the query performance we use two approaches:

- a variation of the Hilbert-packing algorithm [KF93] using the Z-curve instead of the Hilbert-curve for the ordering of the MBBs
- STR - sorted tile recursive [LEL97]

We just present the results of Z-packing here, as we have not observed significant differences in the resulting organization, neither with respect to index size nor to query performance. Moreover, we can state that Z-packing leads to 'better' trees for very skewed data distributions like in our example data sets (this is also mentioned in [LEL97]).

The problem of these packing algorithms is that the chosen clustering scheme is not maintained by the R*-Tree itself. This leads to a fast degeneration of the clustering in the presence of further insertions. The sizes of the different trees have already been presented in Table 5.4: the bulk loaded trees have nearly the same number of data pages, but the different fan-out in the index pages result in large differences for the number of index pages. The results also show the degeneration of the bulk loaded R*-Tree after additional random inserts: while initially the bulk-loaded R*-Tree was up to 5% smaller than the UB-Tree, it is up to 25% larger after random insertion.

5.6 Query performance

The theoretical analysis of query performance is a complex problem as it strongly depends on the data distribution. Cost models for uniform data distribution exist but are not applicable to realistic data sets. There are various approaches [FK94, FSR87, TS96] to analyze the performance of R-Trees and its variants, addressing the problem of overlapping regions. However, all use simplifying assumptions or require detailed information about the data distribution.

As the theoretical analysis is still an open research issue (closely related to the problem of cardinality estimation covered in Chapter 9), we rely on experimental evaluation of the two indexes. To get a somewhat reliable result, we try to cover a large range of data characteristics with our test data sets, including different dimensionality and data distribution.

5.6.1 Point query analysis

We start with the analysis of point queries as these are easier to tackle than range queries, at least for the UB-Tree.

For the UB-Tree the picture is quite clear: one point query requires $K_{UB} = h_{UB} * \log_2 F_{UB}$ key comparisons and $P = h_{UB}$ page accesses ($h_{UB} - 1$ index pages and one data page) for a single point query in best=average=worst case[3].

For the R*-Tree, however, there is a huge difference between best, average, and worst case. In the best case, R*-Trees require only one index page lookup, i.e., $P = 1$, and $K_{RST} = F_{RST}$ key comparisons. In this case, the query has been placed in the empty part of the universe, which the R*-Tree can detect efficiently if no MBB in the root node covers the search point. In some cases, R*-Trees have the same performance as UB-Trees, i.e., exactly one path has to be traversed from the root to a leaf page. The concept of overlapping MBBs, however, causes a drastic degeneration of the performance for point queries in worst case. Instead of following only one path, multiple paths have to be traversed. In worst case, all paths have to be followed; as the degree of overlap depends on the data distribution no theoretical analysis can be given.

In our experiment, we just consider the number of data page accesses. The results in Table 5.6 clearly show that the R*-Tree has to read much more data pages than the UB-Tree, which requires just 1 data page access.

Table 5.6: Data page accesses for point queries on the R*-Tree

Data Set	Index	# Queries	min(P)	avg (P)	max (P)	total P
GEO	RST_40_30	1000	1	2,838	8	2838
	RST_50_0	1000	1	3,117	7	3117
	RST_50_30	1000	1	2,875	7	2875
CENSUS5D	RST_40_30	1000	2	5,642	11	5642
	RST_50_0	1000	2	5,584	10	5584
	RST_50_30	1000	2	5,584	10	5584
GfK3D	RST_40_30	1000	2	5	10	5043
	RST_50_0	1000	2	5	9	5027
	RST_50_30	1000	2	5	10	5042
GfK11D	RST_40_30	1000	1	7,397	35	7397
	RST_50_0	1000	1	8,248	59	8248
	RST_50_30	1000	1	7,791	45	7791

The results indicate that the point query performance of R*-Trees deteriorates with increasing dimensionality.

5.6.2 Range query analysis

For the range query performance, we expect a large difference between the performance on bulk-loaded and on 'random' (i.e., random inserts into bulk-loaded trees) R*-Trees. We, therefore, treat the two cases separately. Also, the performance depends on the size, i.e., volume, and the location of the query box. Consequently,

[3]We assume unique keys, i.e., there are no duplicates in the index.

we classify queries in different volume categories and from each category the queries are randomly chosen, i.e., they have random location and shape.

How to read a box-plot

We use box-plots to visualize the performance results. Instead of showing individual results of a query suite, box-plots summarize the observations (see Figure 5.4 for an example).

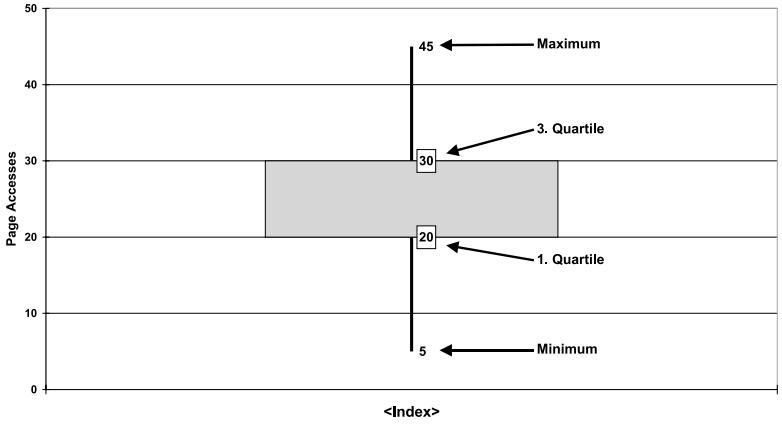


Figure 5.4: Example box-plot

For a set of queries, a box-plot depicts the range between the minimum and the maximum number of data page accesses within the query suite as a **bar**. To get a better understanding of the distribution of the query results, the plot shows also the range of page accesses between the 1. and the 3. quartile according to the required I/O, i.e., excluding the 25% slowest and fasted queries, as a **box** (the width of the box has no meaning, it just eases the presentation). The corresponding values are also presented; the values of the 1. and 3. quartile are framed.

Range queries on the GEO data set

For the GEO data set, the bulk loaded indexes show only minor differences in the performance such that we focus on the complete data set after additional random inserts. The following figures show the box plots for three query classes: first, queries with result sets smaller than 1000 tuples (Figure 5.5); second, queries returning between 1001 and 10.000 tuples (Figure 5.6); third, queries with more than 10.000 tuples (Figure 5.7). The results show that, independent of the query size, all indexes almost show the same performance. It is interesting to note that even for small results the UB-Tree can compete with the R*-Tree for this data set, despite the problem of handling dead space (see Figure 5.5)[4].

[4]For the small queries on the GEO data set, the minimum and the 1. quartile are equal, i.e., we have many queries returning 0 tuples (>25%).

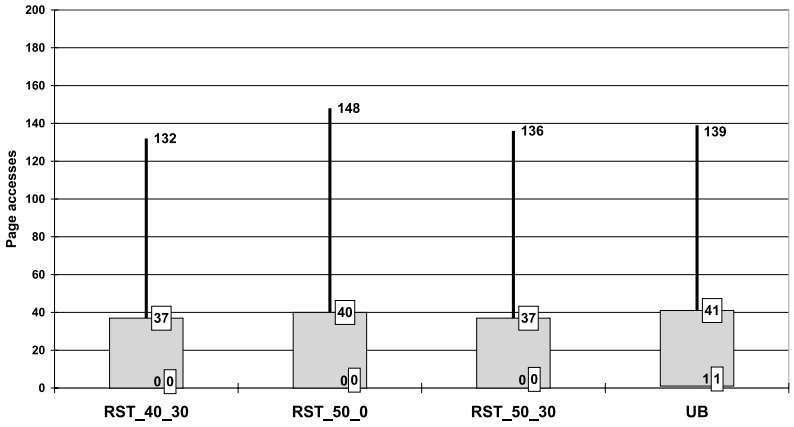


Figure 5.5: GEO [random]: small result set (< 1000 tuples; 697 queries)

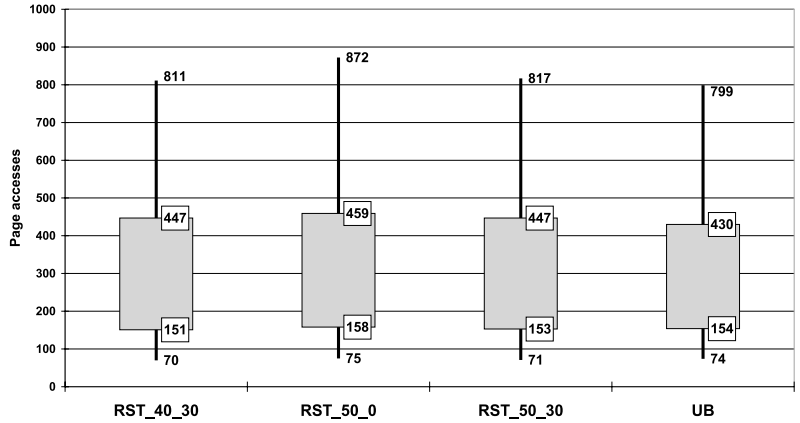


Figure 5.6: GEO [random]: medium result set (1000-10.000 tuples; 505 queries)

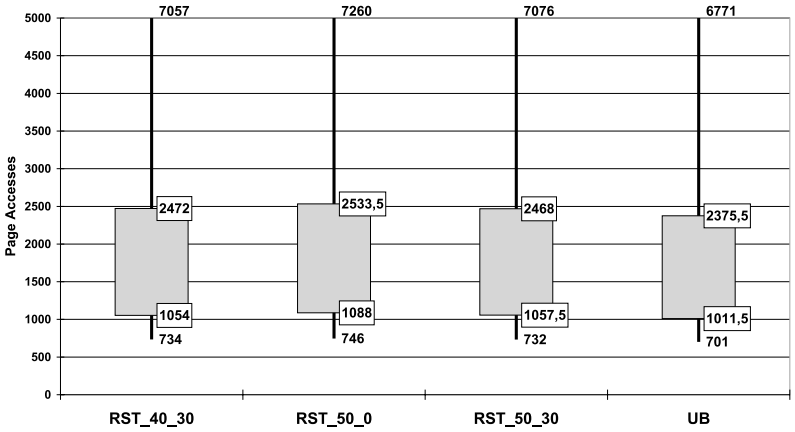


Figure 5.7: GEO [random]: large result set (>10.000 tuples; 343 queries)

Range queries on CENSUS5D

For the CENSUS5D data set, we observe the same behavior as for the GEO data set: while the bulk loaded indexes show the same performance, the R*-Tree performance deteriorates with additional random insertion. Figure 5.8, Figure 5.9, and Figure 5.10 show the performance for the bulk loaded indexes. Here, the R*-Tree shows the better performance for small result sets, where it benefits from reducing the dead space. For larger queries the UB-Tree has a small advantage.

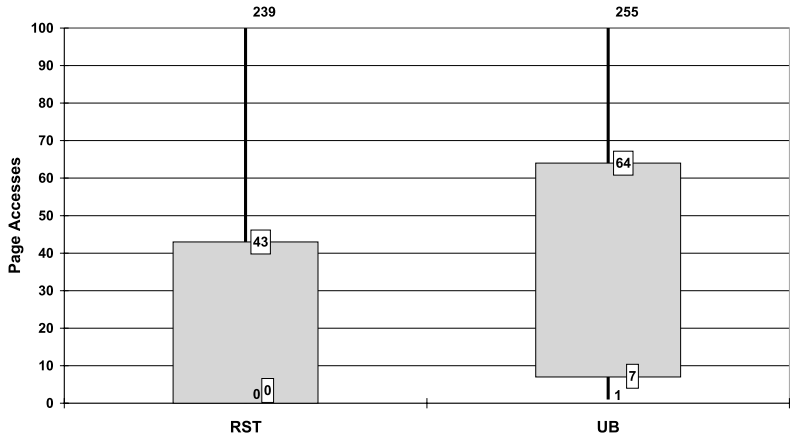


Figure 5.8: CENSUS5D [bulk loaded]: small result set (< 1000 tuples; 440 queries)

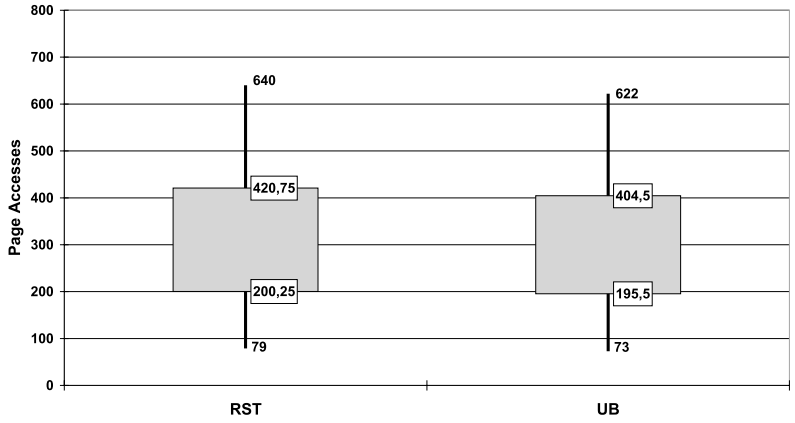


Figure 5.9: CENSUS5D [bulk loaded]: medium result set (1000-10.000 tuples; 453 queries)

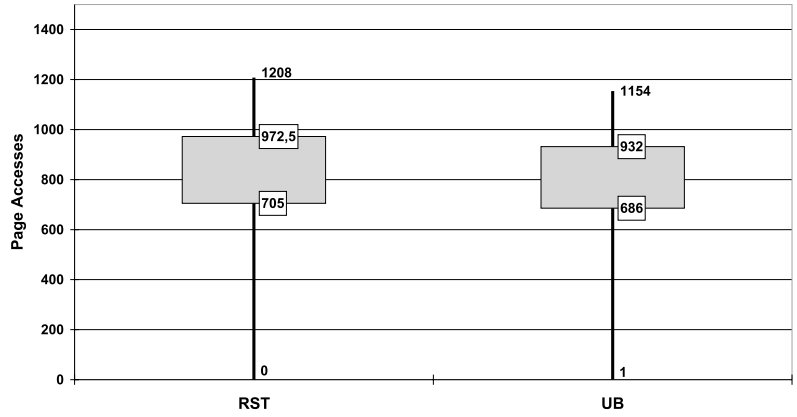


Figure 5.10: CENSUS5D [bulk loaded]: large result set (>10.000 tuples; 107 queries)

The picture changes drastically after random insertions: the UB-Tree now clearly outperforms the R*-Tree in all configurations (Figure 5.11, Figure 5.12, and Figure 5.13). It is interesting to note that the the two R*-Tree configurations with the higher guaranteed minimal page utilization perform better than the one with the lower minimal page utilization.

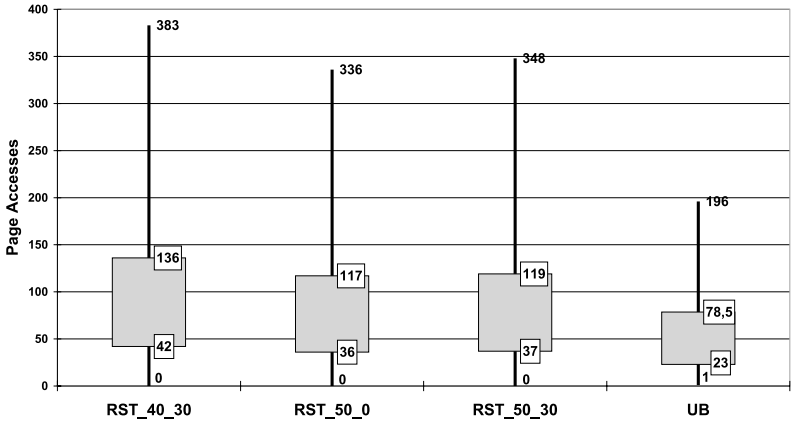


Figure 5.11: CENSUS5D [random]: small result set (< 1000 tuples; 440 queries)

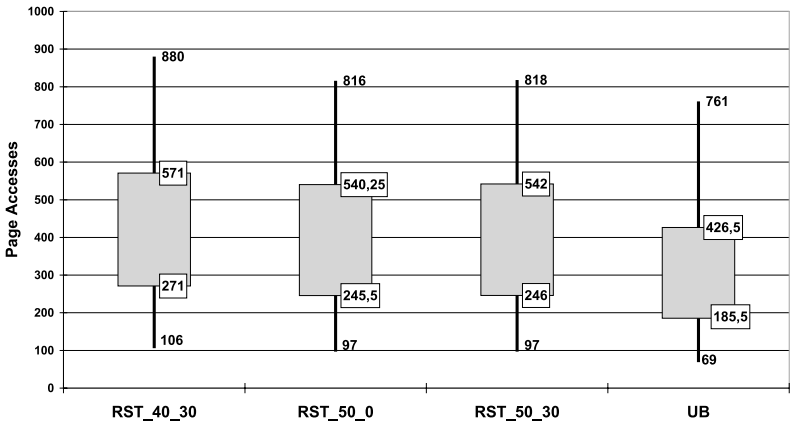


Figure 5.12: CENSUS5D [random]: medium result set (1000-10.000 tuples; 453 queries)

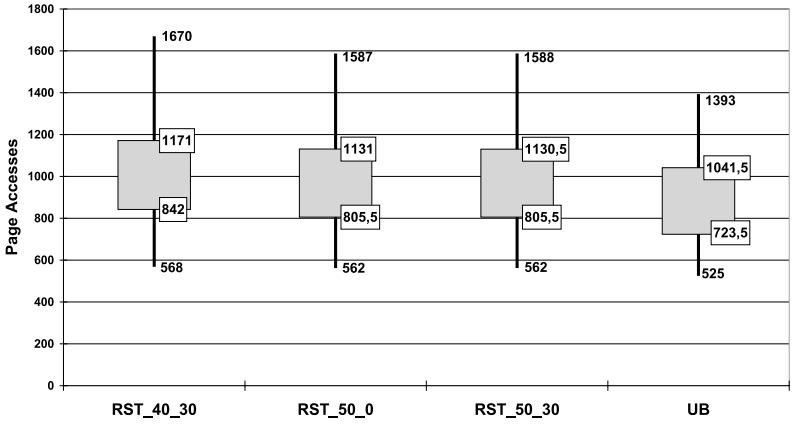


Figure 5.13: CENSUS5D [random]: large result set (>10.000 tuples; 107 queries)

Range queries on GfK3D and GfK11D

For the measurements on the GfK data warehouse we use 604 queries from a typical segmentation report. The queries restrict a specific product group in the Product dimension, a two-month period in the Time dimension, and a country in the Segment dimension. For these queries, we do not distinguish among different result set sizes as the maximal selectivity for all these queries is already below 0,5%.

Figure 5.14 shows the performance of the bulk loaded indexes for GfK3D. Again, there is no major difference as both indexes cluster the data according to the Z-curve. Still, the R*-Tree can benefit slightly from the smaller size.

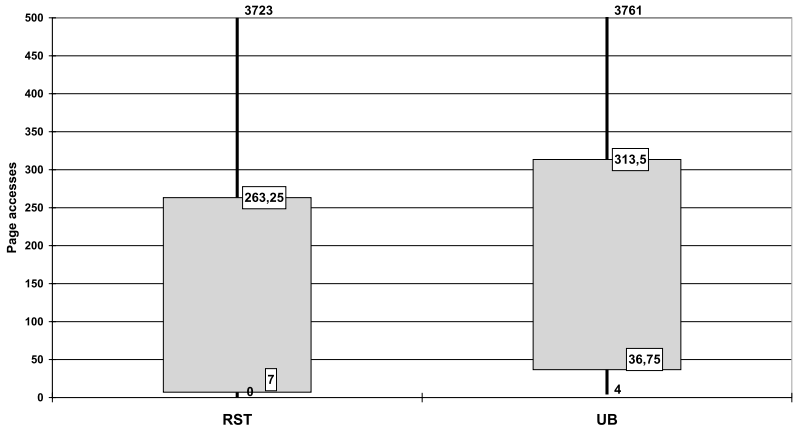


Figure 5.14: GfK3D [bulk loaded]: range query performance

For the data warehouse, the random insertion of new tuples has a different effect than for the other two applications. We add fact data of new time periods, i.e., the data is inserted in not yet occupied space. Consequently, the organization of the R*-Tree is not changed for the old data, leading to the same performance for the queries. However, if we run queries targeting the new data, i.e., changing the Time restriction to the new period, the R*-Tree shows poor performance as the random insertion leads to poor clustering of the new data (see Figure 5.15). For such queries, the UB-Tree requires almost half of the page accesses of the R*-Tree.

The same behavior can be observed for GfK11D in Figure 5.16 and Figure 5.17. For the bulk loaded index, the R*-Tree even has a small advantage over the UB-Tree corresponding from the efficient handling of 'dead space' queries. After random insertion, however, the picture changes drastically: as in the other measurements, the UB-Tree is now the clear winner. The GfK results show the same behavior as for CENSUS5D: the 'optimal' R*-Tree configuration suggested by [BKSS90], i.e., RST_40_30, has the worst range query performance of all three R*-Tree configurations. This is an indication that a larger page utilization is more beneficial for the range query performance than an optimized MBB layout.

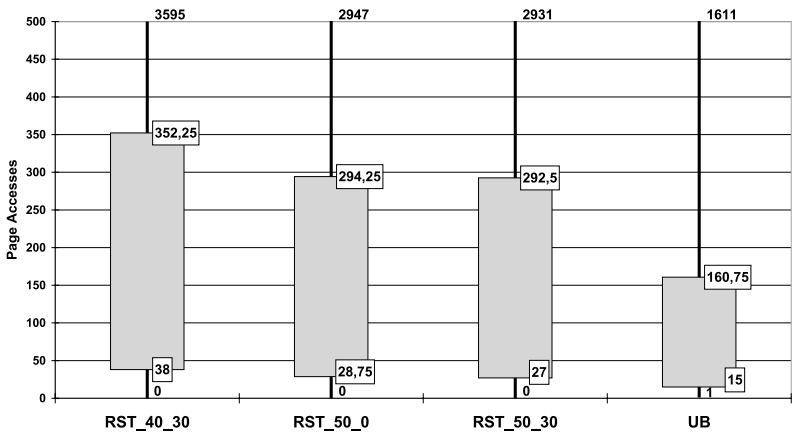


Figure 5.15: GfK3D [random]: range query performance

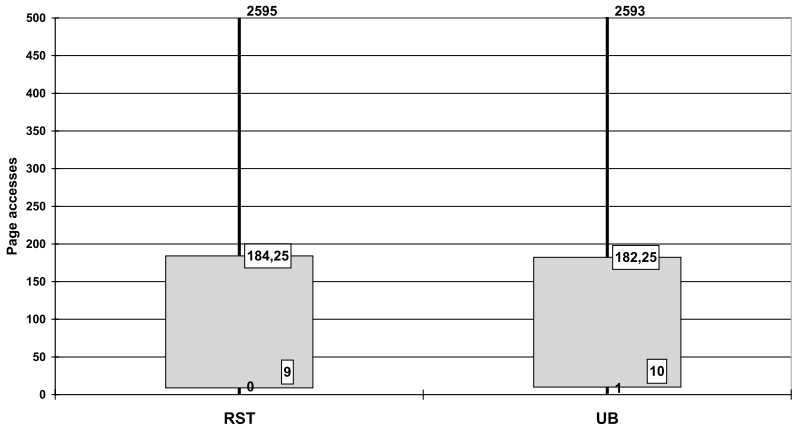


Figure 5.16: GfK11D [bulk loaded]: range query performance

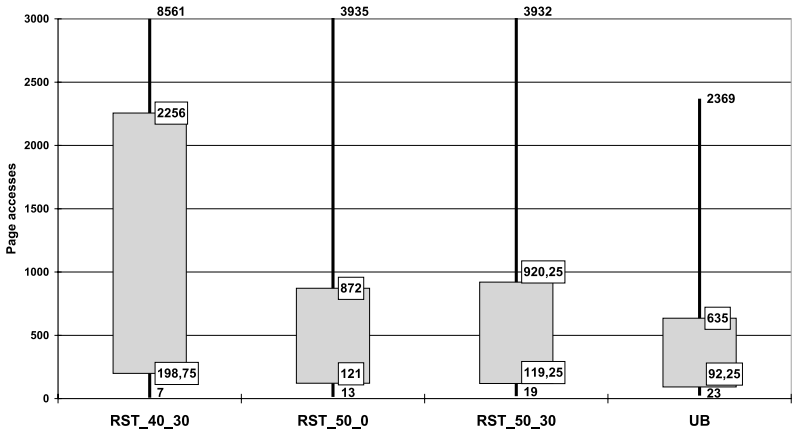


Figure 5.17: GfK11D [random]: range query performance

5.7 Chapter notes and related work

In this chapter, we have addressed the first objective of this work, the search for a general-purpose, multidimensional index structure. Before different data structures can be evaluated, we require a solid framework that allows for a fair comparison. To this end, we define a framework that does not only consider query performance but also takes important practical issues like space consumption or multi-user behavior into account.

We present the results of a detailed theoretical and experimental comparison of R*-Trees and UB-Trees, which we regard as the most promising candidates out of the vast set of proposed multidimensional indexes.

More specifically, we can state that R*-Trees are in general significantly larger (i.e., occupy more disk pages) than UB-Trees. This directly leads to higher maintenance costs, which are even further increased by the concept of reinsertions. More than two times higher maintenance costs go hand in hand with reduced concurrency, making the R*-Tree inapplicable for dynamic applications.

Regarding query performance, the UB-Tree outperforms the R*-Tree for point and range queries. Only so-called 'dead-space' queries, i.e., queries into the empty part of the universe, are usually processed faster by the R*-Tree than by the UB-Tree. The difference in query performance is not only the result of the differences in the index size, but is also based on the superior clustering achieved by the Z-curve. The clustering of the UB-Tree proves to be more robust against various data distributions and increasing dimensionality.

The results of this comparison show that the UB-Tree comes much closer to the notion of a general access method for multidimensional point data than the R*-Tree. It combines efficient query processing with modest space requirements and reasonable maintenance performance. Furthermore, the underlying B-Tree technology provides the required concurrency control methods and significantly eases the integration into existing systems as we will show in the next part of this thesis.

Related Work

Comparison of data structures is an inherent part of the proposal of any new access method. Therefore, a large body of work about comparing indexes is available. [GG98] provides an excellent overview on most of the important multidimensional access methods and contains some qualitative performance results. However, most publications only consider the query performance and not the whole range of important criteria for indexes. The first publication to provide a more general comparison framework is [KSSS90]. It does not only consider the index size and the insertion performance, but also takes various data distributions and real world data into account. Still, this work lacks in including the CPU cost into the comparison. [ZMR96] describe general guidelines for the comparisons of data structures.

There are also many contributions towards an analytical evaluation of different data structures. Especially the range query performance of R-Trees and its variants has been the focus of [TS96, FSR87]. The general problem of the analytical models is to find a way to capture non-uniform and correlated data, in order to derive

models suitable to address real-world data distributions. A promising approach is the concept of 'fractal dimensions' by [FK94].

Comparison to other data structures In [Jür99] Bitmap indexes and R*-Trees are investigated w.r. to their usability in data warehouses. Furthermore, an extension of the R-Tree, the R_a^*-Tree has been proposed to speed up aggregations queries. The comparison of the various data structures is based on simulation using a cost model and shows that the R_a^*-Tree outperforms the standard R*-Tree for aggregation queries. An additional result of this work is that Bitmap indexes outperform tree-based data structures for higher (i.e., $d \geq 4$) dimensions. However, the comparison only covers non-clustering indexes such that results can not be applied to compare clustering indexes and Bitmap indexes.

Part II

Integration and optimization of UB-Trees

As shown in the first part, the UB-Tree comes very close to the notion of a general-purpose multidimensional access method w.r. to performance and flexibility. In this part, we discuss the underlying concepts of the UB-Tree in more detail. We demonstrate the facile integration into a DMBS kernel and discuss optimizations increasing the functionality and efficiency.

Chapter 6

Integration of the UB-Tree into a DBMS kernel

Integrating a new access method into a DBMS kernel is usually a very costly task. Besides integrating the algorithms of the new access method itself, much effort has to be spent to provide locking and logging mechanisms for full transactional functionality in multi-user environments. In comparison to other multidimensional access methods, the UB-Tree has the advantage to be based on the standard B-Tree, which is already available in almost all DBMSs. Consequently, one can rely on the existing implementations, including proven locking and logging protocols developed for B-Trees. Thus, the integration effort reduces to the integration of the UB-Tree specific algorithms and the corresponding modification of the query processor. We will highlight some of the implementation details of the UB-Tree algorithms and the key issues to be solved during an integration.

6.1 The UB-Tree: Advanced concepts

In Section 5.2.2, we introduced the UB-Tree as the extension of the standard B-Tree to multiple dimensions. It uses the space-filling Z-curve to achieve a linearization of the multidimensional universe. The linear space is then indexed by a standard B-Tree. A UB-Tree can be regarded as a special version of the B-Tree with computed keys, using the linearization of the Z-curve as key function. We now introduce the UB-Tree more formally.

6.1.1 Z-regions

As mentioned earlier, the concept of *Z-regions* is fundamental for the UB-Tree.

Let Ω be the d-dimensional universe with domains $\mathbb{D}_1, \ldots, \mathbb{D}_d$ having the same length l for their binary representation[1], i.e., $\Omega = \mathbb{D}_1 \times \ldots \times \mathbb{D}_d$. Let $\mathcal{Z} \subset \mathbb{N}_0$ be the domain of all Z-values for Ω, i.e., $|\mathcal{Z}| = |\Omega|$.

[1]We use the restriction only for the ease of illustration. The address calculation can be easily extended to varying domains with different lengths.

Definition 6.1 (Z-value, Z-address)
*For $x \in \Omega$ and the binary representation of each attribute $x_i = x_{i_{l-1}} x_{i_{l-2}} \dots x_{i_0}$ we define the **Z-value** (or **Z-address**) $Z(x) : \Omega \mapsto \mathcal{Z}$ as*

$$Z(x) = \sum_{j=0}^{l-1} \sum_{i=1}^{d} x_{i_j} \cdot 2^{j \cdot d + i - 1}$$

Without proof we state that $Z(x)$ is a bijective function with the inverse function $Z^{-1} : \mathcal{Z} \mapsto \Omega$ as:

$$Z^{-1}(\alpha) = x = (x_1, \dots, x_d) \text{ with } x_i = \sum_{j=0}^{l-1} \alpha_{j \cdot d + i - 1} \cdot 2^j$$

For a Z-value $z \in \mathcal{Z}$, z^d denotes the bit-representation of dimension d. For a bit z_i of a Z-value $z \in \mathcal{Z}$, $dim(z, i)$ returns the dimension the bit i corresponds to; we sometimes just write $dim(i)$.

We use three different representations of Z-values:

1. Integer value, denoting the position on the Z-curve
2. Bit-String, representing the integer value
3. *Step-Representation*: by combining all bits of the same step to one integer value one gets a compressed "binary" representation. The step denotes the position of the bit beginning with the highest-valued bit in each dimension. For better readability the integers representing one step are delimited by '.'.

Example 6.1: Z-value representation

Let $\mathbb{D} = \{x | 0 \leq x \leq 7\}$, and $\Omega = \mathbb{D} \times \mathbb{D}$. Consequently, the length of the binary representation is $l = 3$. For $x = (0, 1) \in \Omega$, $Z(x) = 2 = 000010 = 0.0.2$; for $y = (3, 2) \in \Omega$, $Z(y) = 13 = 001101 = 0.3.1$.

◇

Based on the concept of Z-values we now can define Z-regions.

Definition 6.2 (Z-region)
*A **Z-region** $[\alpha, \beta]$ is the space covered by the interval on the Z-curve defined by the two Z-addresses α and β, i.e., $[\alpha, \beta] = \{x \in \Omega | \alpha \leq Z(x) \leq \beta\}$.*

Given Definition 6.2, we can treat a Z-region either as a one-dimensional interval of Z-values or a shape in multidimensional space representing the space covered by the Z-curve. Consequently, we can use Z-regions to define a complete partitioning of the multidimensional universe.

Definition 6.3 (Z-region partitioning)
*A set of Z-regions $R = \{R_1, \ldots, R_n\}$ is called **Z-region partitioning of** Ω, iff*

1. $\forall i, j : R_i \cap R_j = \emptyset$
2. $\bigcup_{i=1}^{n} R_i = \Omega$

For a Z-region partitioning R either the start address α_i or the end address β_i of the Z-region R_i is determined by the previous Z-region or the following Z-region in Z-order resp. (see Example 6.2). Therefore, we only use the end address for the unique identification of a Z-region and call β_i the **Z-region address** (or short **address**) of Z-region R_i.

For completeness, we state the connection theorem for Z-regions, the proof can be found in [Mar99]:

Lemma 6.1 (connection of Z-regions)
Any Z-region consists of at most two spatially disconnected sets of points and such Z-regions exist.

Lemma 6.1 holds for all universes Ω, independent of the dimensionality d of Ω.

For the definition of the UB-Tree we finally have to specify the mapping between Z-regions and physical pages.

Definition 6.4 (Z-regions and pages)
A page P corresponds to a Z-region R (denoted by $P \leftrightarrow R$), iff P contains exactly those tuples belonging to R, i.e., $P \subseteq R$.

Definition 6.5 (UB-Tree)
*A **UB-Tree** is any variant of a B-Tree with $Z(x)$ as address function.*

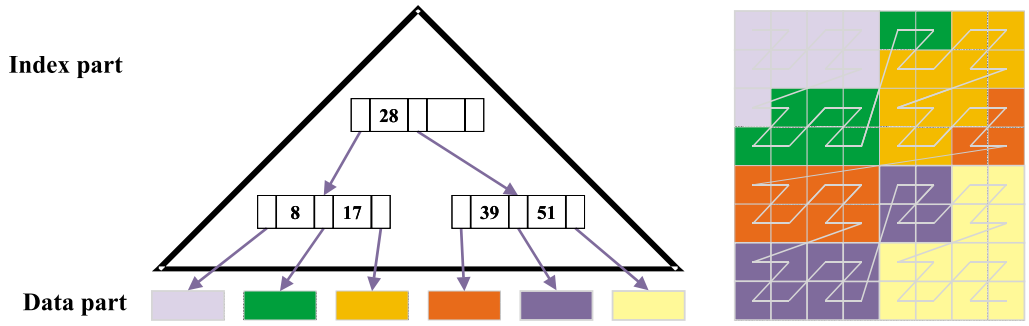
Definition 6.5 directly leads to the following important properties of UB-Trees:

- each page of the underlying B-Tree specifies an interval on $\mathcal{Z}$ and thus defines/corresponds to a Z-region of the UB-Tree. We will therefore use the terms Z-region and page synonymously.
- all pages of the B-Tree consequently constitute a Z-region partitioning of the universe.

Example 6.2: UB-Tree concepts

Figure 6.1 illustrates the various concepts of the UB-Tree. Figure 6.1(a) shows the B-Tree with the Z-values as addresses. This UB-Tree partitions the universe Ω into six Z-regions, namely $R_1 = [0, 8], R_2 = [9, 17], R_3 = [18, 28], R_4 = [29, 39], R_5 = [40, 51]$, and $R_6 = [52, 63]$. Figure 6.1(b) shows the corresponding Z-region partitioning of the two-dimensional universe.

$\diamond$



(a) B-Tree with Z-values

(b) Z-region Partitioning

Figure 6.1: The UB-Tree: Example B-Tree and resulting Z-region partitioning

6.1.2 Z-Ordering and bit-interleaving

An important issue of UB-Trees is the representation and the computation of the addresses, i.e., the Z-values of the tuples. Z-values are efficiently computed by bit-interleaving [OM84, TH81]. In order to apply bit-interleaving, however, we require an order-preserving binary representation of the attribute values. With respect to implementation, Z-values are represented as variable length[2] *bitstrings*, i.e., a sequence of bits.

Bit-interleaving and data type transformation

To compute the ordinal number of a tuple on the Z-curve, bit-interleaving is applied. That is, all the bits of the binary representations of the index attributes are interleaved in an arbitrary, but fixed, order (see Figure 6.2).

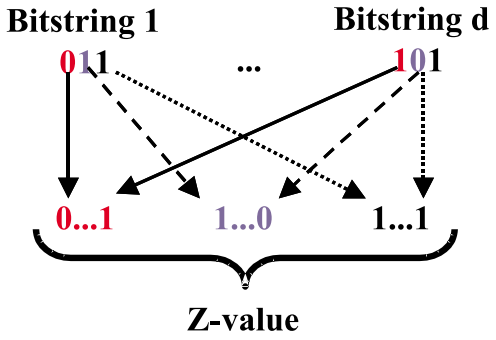


Figure 6.2: Z-value computation based on bit-interleaving

In order to compute correct Z-values, bit-interleaving requires the bit representations of the index attributes to adhere to the natural order on the corresponding

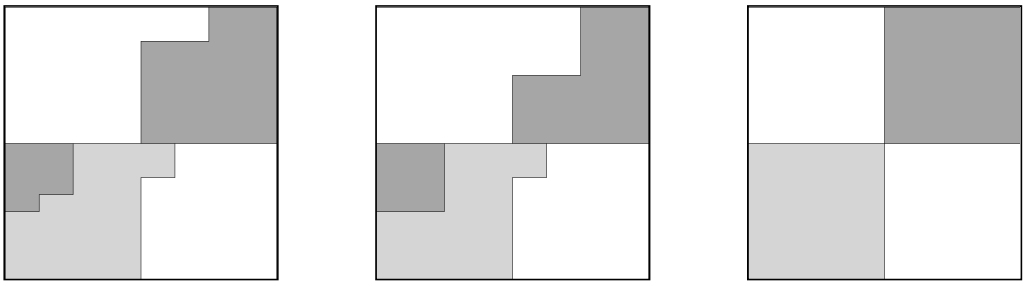
[2]For a given UB-Tree instance the length is fixed.

domains. For unsigned integers, for example, the given binary representation suffices. For signed integers, however, a transformation function is needed to take care of the sign bit, which causes negative integers to be greater than positive integers with respect to the bit-lexicographic order on the binary representation. Consequently, for each data type to be supported by UB-Trees, an appropriate transformation function has to be specified. The transformation functions can also be used for allowing for more powerful semantics, e.g., SOUNDEX codes or case insensitive search on strings. If possible, it is useful to normalize attributes to unsigned integers starting with 0 as one step of the transformation, as this allows for a much better space partitioning with shorter Z-values. One example for normalization for complex data types is the hierarchy encoding presented in Section 2.5. Other possibilities for normalization include hashing or more complex non-linear methods. Note that complex normalization may lead to significant performance overhead, which are no longer neglectable. In such cases, it may be worthwhile to store the index keys together with the tuples on the leaf pages in order to reduce the CPU cost. Our standard normalization techniques just require a few microseconds of CPU time, and therefore do not affect the address calculation performance.

6.1.3 Basic functionality: insertion, deletion, update, and point search

The basic operations on UB-Trees are directly mapped to the underlying B-Tree. To perform an insertion, deletion, or update the Z-value for the corresponding tuple is computed in the first step. This Z-value is then used as the address in the operations on the B-Tree. The same holds for point queries. As consequence, the performance guarantees for B-Trees also apply to the basic operations of UB-Trees. Only with respect to page splitting, the UB-Tree algorithm differs from the standard B-Tree algorithm. The calculation of the page separator is adapted to achieve a better space partitioning, i.e., creating rectangular regions whenever possible. This influences the range query performance by reducing the number of regions overlapped by a range query. This can be achieved by choosing the shortest Z-value (i.e., the Z-value that has as many trailing zero bits as possible) between the two middle tuples s and t as new separator, instead of s, t, or another Z-value in the middle of the page. This modification adds no complexity to the split cost as it is done in $O(l)$ bit operations, where l is the length of the Z-value in bits, and the worst case page utilization of 50% is still guaranteed [Mar99]. The advantage of this split strategy is twofold: first, better range query performance on average. Secondly, shorter separators lead to a more compact index part of the UB-Tree, a phenomenon also exploited in Prefix-B-Trees [BU77].

A further improvement of the space partitioning is achieved by relaxing the minimum page-filling guarantee. The ϵ-Split algorithm looks for the shortest Z-value in the range of ϵ% around the middle of the page. This reduces the minimum page utilization to $(50 - \epsilon)$% but leads to a better space partitioning, i.e., to a reduction of fringes. Figure 6.3 illustrates the improved space partitioning: Figure 6.3(a) shows a 2-dimensional partitioning resulting from the standard split algorithm. In

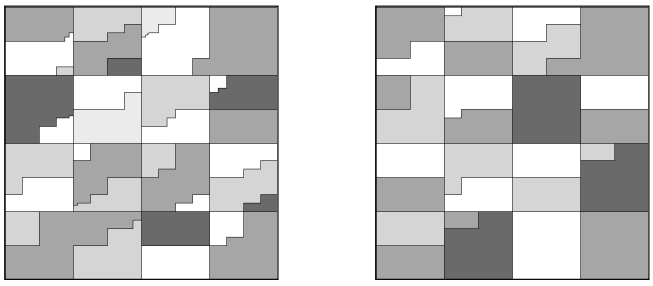


(a) w/o ϵ-Split (b) with ϵ-Split (ϵ=14%) (c) "ideal split"

Figure 6.3: Space partitioning and split strategy for a 2-dimensional universe

Figure 6.3(b), the ϵ-Split leads to more rectangular regions, which are closer to the ideal case depicted in Figure 6.3(c).

Figure 6.4(a) and Figure 6.4(b) picture the influence of the ϵ-Split for a larger database.



(a) standard split (b) with ϵ-Split (ϵ =14%)

Figure 6.4: Effect of ϵ-Split for a larger database

The benefit of the optimized space partitioning caused by ϵ-splitting is an improved query performance as the number of fringes is reduced, thus leading to less pages unnecessarily intersected by a query box (details of the range query algorithm are covered in the next section).

The optimal choice of ϵ is a tradeoff between storage utilization and quality of space partitioning: the higher ϵ, the better the space partitioning, and as consequence the range query performance, but the lower the worst-case storage guarantee. Figure 6.5 shows the results of 6-dimensional range queries with varying volume and location on a 6-dimensional UB-Tree for growing ϵ. The empirical results show that already a small ϵ (around 5%) leads to significant improvement of the space partitioning. For $\epsilon > 15\%$ no further improvements in the space partitioning are observed. It is important to note that the ϵ-Split has the same complexity as the

regular split algorithm.

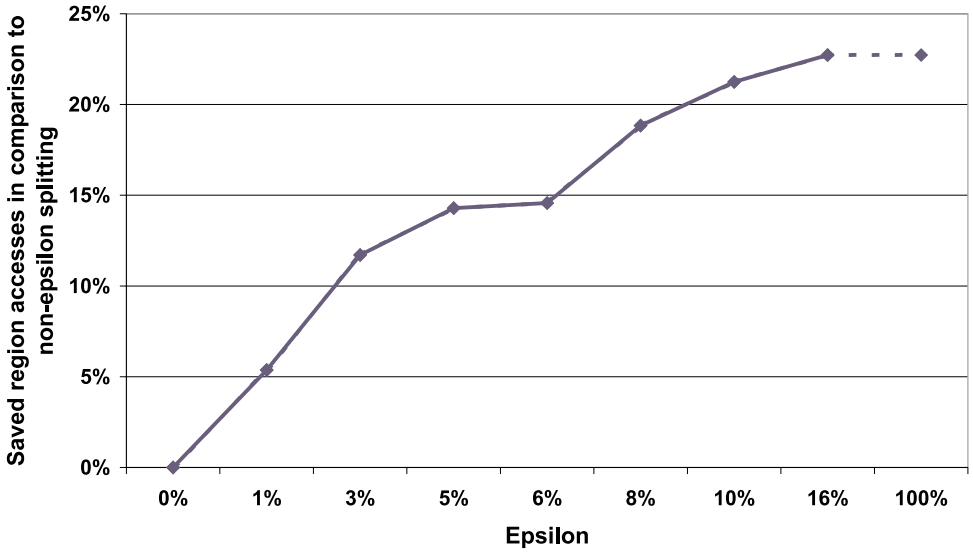


Figure 6.5: Trade-off between reduced page utilization and improved query performance

6.2 The UB-Tree range query algorithm

The goal of processing multidimensional range queries is to minimize the required I/O. As consequence, the UB-Tree should only retrieve Z-regions, i.e., pages from secondary storage, that may contain result tuples. This results in retrieving only those pages corresponding to those Z-regions, which are properly intersected by the query box. Due to the mapping of the multidimensional space to Z-values, a query box in multidimensional space partitions into a set of intervals on the Z-curve, called *Z-intervals*. Figure 6.6 shows such a decomposition for a query box.

Due to the nature of the Z-curve, the decomposition may lead to a large set of intervals, many of which may be located in the same Z-region. This will cause multiple accesses to the same page or additional overhead to prevent these accesses. For processing the range query, however, it is sufficient to determine the intersected Z-regions, if the Z-regions are post-filtered with the query predicate after retrieval.

Following this idea, the resulting iterative range query algorithm works as follows. Let the multidimensional range restriction be specified by a query box QB with a starting corner and an ending corner, which are given by the two tuples ql and qh ($\mathrm{Z}(ql) \leq \mathrm{Z}(qh)$), respectively. In a first step, the algorithm computes the Z-values for ql and qh, then the region containing ql is located[3]. The range query algorithm then iteratively determines all the regions intersected by the query box

[3]The tuples ql and qh do not have to exist in the UB-Tree. It is sufficient to locate the Z-region, which contains ql.

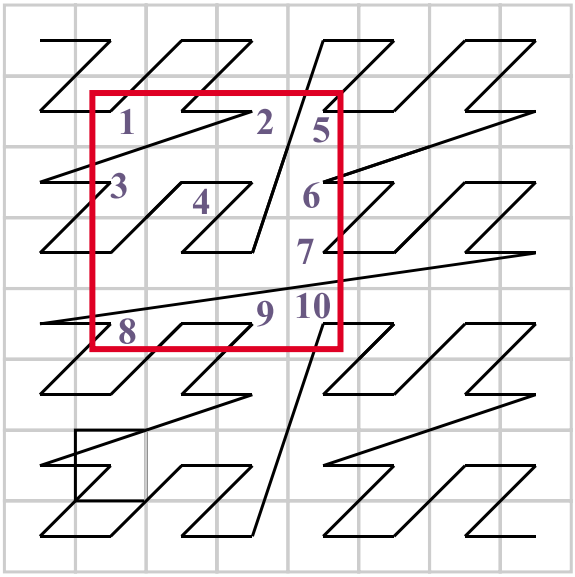


Figure 6.6: Linearization of a query box

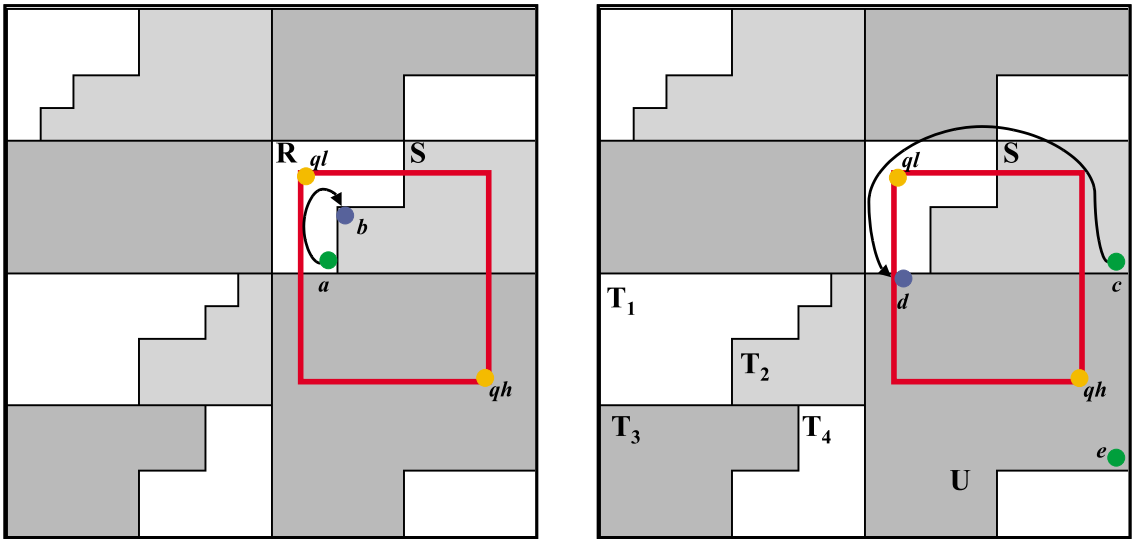
by calculating the Z-value for the next intersection point of the Z-curve with the query box based on the currently processed region/page. This is continued until a region with a region address larger or equal than Z(qh) is retrieved. The algorithm is given as pseudo-code in Figure 6.7 and is visualized in Figure 6.8.

```
Status UBTree_Range_Query(Tuple ql, Tuple qh) {
   Z-value start=Z(ql);
   Z-value end = Z(qh);
   Z-value cur = start;
   //continue as long as we are in the query box
   while (1) {
      //getting the address of the region containing cur
      cur = getRegionSeparator(cur);
      //post-filtering of the tuples in the region
      postFilterPage(GetPage(cur), ql, qh);
      //stop once we cover the whole query box
      if ( cur >= end) break;
      //calculation of next region
      cur = getNextJumpIn(&cur, ql, qh);
   }
}
```

Figure 6.7: The UB-Tree range query algorithm

Figure 6.8 illustrates the algorithm with a small example. In the first step (see Figure 6.8(a)), region R that contains ql is located and post-filtered. Using the region address a of R the next intersection point is calculated. This results in Z-address $b = a + 1$ belonging to Z-region S. The end address c of S, however, is not contained in the query box (cf. Figure 6.8(b)). The algorithm calculates d of Z-region U as next point on the Z-curve to be inside the query box, and continues with processing of U. The four Z-regions $T_1, ..., T_4$, which lie between S and U are

skipped. After processing U the algorithm terminates as the end address e is larger than the upper end qh of the query box.



(a) Jump within query box

(b) Jump into query box

Figure 6.8: Range query processing

All these steps are performed in O(l) bit operations where l is the length of the Z-value. The separator computation and the loading of the page is performed by one B-Tree access. After post-filtering all matching tuples of the fetched page, the tuples are already returned to the caller allowing for pipelining by the DBMS.

6.2.1 Calculating the next intersection point

Calculating the 'next intersection point' (also called 'next-jump-in' (NJI)) of the query box with the Z-curve is the crucial part of the UB-Tree range query algorithm. In the following we will show that this step only requires bit operations on Z-values and no I/O or B-Tree search is necessary. Starting point for this algorithm is the region address cur of the current region. The task is to find the next intersection point of the Z-Curve with the query box Q. This next-jump-in nji is the minimal Z-value larger than the current region address and which is inside Q, i.e., $nji = \min\{y \in \mathcal{Z} | y > cur \wedge Z^{-1}(y) \in Q\}$.

We describe a version of the algorithm that is not optimized for the Z-curve, but works for more general cases as we will discuss in Chapter 10. Let the query box Q be specified by the two Z-addresses min and max. The current address cur is not in Q, i.e., $Z^{-1}(cur) \notin Q$, and $cur < max$, if $cur > max$ then the query box is already completely processed.

For the explanation of the algorithm we need to introduce the notion of a *violation*: a violation is a bit position v in a Z-address $z \in Q$ that causes z' to violate Q, i.e., $z' \notin Q$, if z' is created from z by changing the bit at position v, and possibly bits

at positions smaller than v. A formal definition of violations is given in Chapter 7, Definition 7.2.

Analyzing *cur*

We start with comparing *cur* bitwise with min and max and obtain the following information:

- the highest bit position v in *cur* that is part of a violation of Q
- for each dimension i: bit position $gtMin_i$, which specifies from which bit position on cur^i is greater than min^i; $gtMin_i = -1$ if $cur^i \leq min^i$
- for each dimension i: bit position $ltMax_i$, which specifies from which bit position on cur^i is smaller than max^i; $ltMax_i = -1$ if $cur^i \geq max^i$
- bit position z larger than v that can be safely set to 1 without violating the query box:
$$z = min\{x \geq v | cur'_x = 0 \wedge ltMax_{dim(x)} > x\}$$
If no such position z can be found, the assumption $cur < max$ is violated.

Manipulation of *cur*

With this information we can manipulate *cur* to get the next-jump-in. As $nji > cur$ has to hold, we first have to find a bit in *cur* that we can set to 1. With bit position z we already have found the correct bit: it guarantees the smallest possible enlargement of *cur*. At the same time, no bits at positions larger than z will be changed, i.e., the prefix of length $|cur| - z + 1$ is the same for *cur* and nji. With setting $cur_z = 1$ we may jump too far, i.e., we get an address that is larger than the actual next-jump-in and we have not corrected all query box violations, yet. Consequently, the bits at positions smaller than z have to be changed. This is done for each dimension independently: if the dimension i is already guaranteed to exceed the minimum based on the value of the prefix, i.e., if $gtMin_i > z$, then all bits of this dimension at positions smaller than z can be set to 0. Otherwise, the dimension is set to the minimum of the query box. Formally, nji is specified as

$$\begin{aligned} \forall i > z : nji_i &= cur_i \quad \wedge \\ nji_z &= 1 \quad \wedge \\ \forall i < z : nji_i &= \begin{cases} 0, & \text{if} \quad gtMin_{dim(i)} > z \\ min_i, & \text{else} \end{cases} \end{aligned}$$

Example 6.3: Next-Jump-In algorithm

We use a two-dimensional, 8×8 universe for our illustration of the algorithm, i.e., $\Omega = \mathbb{D} \times \mathbb{D}$, with $\mathbb{D} = \{x | 0 \leq x \leq 7\}$.

We consider the query box $Q = [[min, max]]$ with $min = 0.2.3 = 001011$ and $max = 3.0.2 = 110010$ (see Figure 6.9).

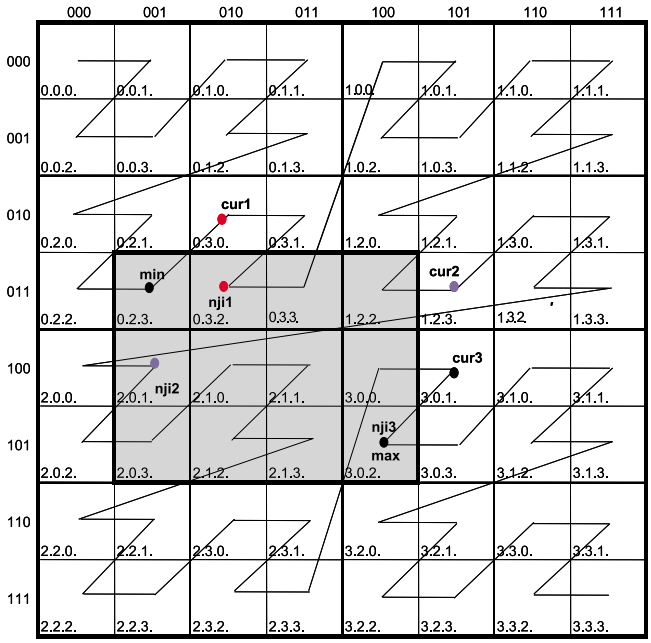


Figure 6.9: Query box Q with three NJI examples

- 1. Example: $cur1 = 0.3.0 = 001100$
  The comparison of $cur1$ with min and max results in:
  $gtMin_1 = 2$ and $gtMin_2 = -1$, and a violation of the minimum in dimension 2: $v = 1$, $z = 1$
  Applying the manipulation routine results in $nji1 = 001110 = 0.3.2$

- 2. Example: $cur2 = 1.2.3 = 011011$
  The comparison with min and max results in:
  $gtMin_1 = 4$ and $gtMin_2 = -1$; there is a maximum violation in dimension 1: $v = 0$, $z = 5$
  Applying the manipulation routine results in $nji2 = 100001 = 2.0.1$

- 3. Example: $cur3 = 3.0.1 = 110001$
  The comparison with min and max results in:
  $gtMin_1 = 4$ and $gtMin_2 = 5$; there is a maximum violation in dimension 1: $v = 0$, $z = 1$
  Applying the manipulation routine results in $nji3 = 110010 = 3.0.2$

◇

6.2.2 Implementation and complexity

We do not present the complete NJI algorithm in pseudo-code as the basic procedure should be clear from the description above. There is just a minor modification necessary for NJI in order to work with the range query algorithm of Figure 6.7. For the NJI we assume that $cur \notin Q$, but in the range query algorithm this is not checked. Following variation of NJI guarantees a correct processing: we first increment cur by one, i.e., we create $cur' = cur+1$. We then perform the comparison of cur' with min and max to derive the required information. If the comparison shows that $cur' \in Q$ then we have already found the next-jump-in, i.e., $nji = cur'$, otherwise we perform the required manipulation.

For the complexity of the algorithm it is important to note that no I/O operations are required, but just bit operations on Z-addresses. For a Z-address length of l, the NJI algorithm requires:

$$\underbrace{3 * l}_{\text{comparison of cur and min}} + \underbrace{3 * l}_{\text{comparison of cur and max}} + \underbrace{l}_{\text{manipulation of cur}} = 7 * l$$

in worst case. Often the comparison can be terminated earlier, i.e., as soon as $gtMin_i$, $ltMax_i$, and v are determined.

6.2.3 Proof of correctness

To prove the correctness of the next-jump-in (NJI) algorithm we have to show the following things:

1. $nji \in Q$
2. $\forall \alpha : cur \leq \alpha < nji : \alpha \notin Q$; we assume without loss of generality that $cur \notin Q$ holds.

1. holds due to construction of nji: all bits that violate the query box are either set to 0 or to the corresponding value of the minimum thereby guaranteeing that the minimum in each dimension is not violated.

2. Assume $\exists \alpha : cur < \alpha < nji$ with $\alpha \in Q$
   $\forall i > z : cur_i = nji_i \Rightarrow \forall i > z : \alpha_i = cur_i$
   $\alpha < nji \Leftrightarrow \exists q : \alpha_q = 0 \wedge nji_q = 1$

   We distinguish two cases for q:

   (a) $q < z : \quad nji_q = 1 \overset{\text{NJI-Algo}}{\Longleftrightarrow} nji^{dim(q)} = min^{dim(q)}$
   $\Rightarrow \alpha^{dim(q)} < min^{dim(q)} \rightarrow$ contradiction to $\alpha \in Q$

   (b) $q = z : \quad \alpha > cur \Leftrightarrow \exists p < z : \alpha_p = 1 \wedge cur_p = 0$
   z is minimal save bit $\Rightarrow \alpha^{dim(p)} > max^{dim(p)} \rightarrow$ contradiction to $\alpha \in Q$

□

6.3 Advanced query algorithms

During the MISTRAL project, various other query algorithms have been developed for the UB-Tree. Very relevant in practice is the extension of the basic range query algorithm to handle multiple query boxes at the same time [FMB99]. Handling multiple query boxes simultaneously allows for avoiding accessing overlapped regions (i.e., pages) multiple times, and thus reduces processing time in comparison to sequential processing significantly. Processing of multiple query boxes is for example required for handling query predicates with conjunctions (e.g., OR terms in the where clause).

For sorted processing of a query box, the Tetris algorithm [MZB99] avoids costly external sorting for large results sets by utilizing the partitioning created by the UB-Tree in a kind of sweepline-fashion. In [ZMB01] the same concept is used in the TempTris algorithm to create a UB-Tree efficiently out of an already sorted stream of tuples. Both algorithms and their applications are described and investigated in detail in [Zir02].

Nearest neighbor (NN) queries play an increasingly important role in new applications. In [Mar99] two NN-algorithms are proposed for UB-Trees.

6.4 The integration project

In the ESPRIT project MDA funded by the European Commission the UB-Tree has been integrated into the commercial DBMS TransBase of TransAction Software GmbH, Munich. TransBase is a full-scale, client-server architecture, relational database system, which conforms to the SQL-92 standard.

The most important prerequisite for the integration is the existence of a clustering B-Tree. With that, the integration reduces to the following tasks:

- Extension of Data Definition Language (DDL)
- Extension of query optimizer and query processor
- Integration of UB-Tree algorithms

Figure 6.10 shows the changes of the individual database kernel modules required by the UB-Tree integration. The shaded boxes mark the modifications in the single modules, where darker shading signals the more complex modifications.

DDL extension

The goal of the integration is to be as transparent to the user as possible. With respect to the query language this is not a problem as the UB-Tree is treated as any other access method of the DBMS. Obviously, the DDL has to be enhanced to allow for the creation of UB-Trees. For this purpose, various DDL statements, e.g., CREATE TABLE or CREATE INDEX, are extended. To handle the new index structure, minor additions to the system catalog are required. Most importantly,

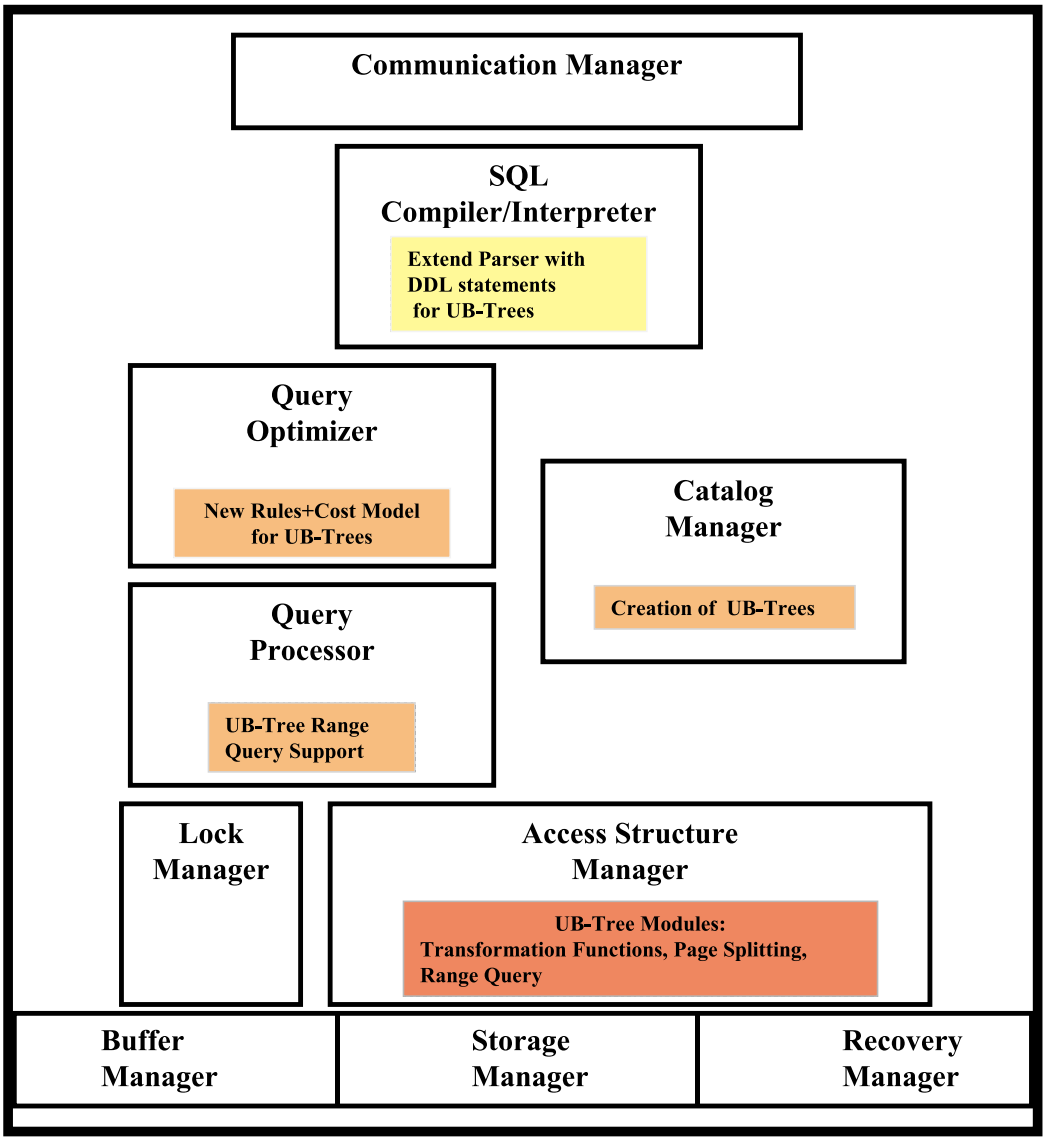


Figure 6.10: Overview of kernel modifications required for UB-Tree integration

information for the transformation functions of the indexed attributes has to be stored.

Query optimizer and processor extension

For getting the greatest benefits out of the new index, the adaption of the query optimizer and the query processor is of greatest importance. We deal with the optimizer issues specifically in Chapter 9. The query processor has to be changed in order to support the query algorithms of the UB-Tree. It depends on the architecture of the target DBMS whether these issues are solved on the level of the access method or in the query processor.

UB-Tree algorithms

Finally, the UB-Tree algorithms have to be integrated with the existing B-Tree functionality. Depending on the extensibility of the existing B-Tree, this task requires only minor changes to existing code.

6.5 Performance evaluation of the integrated UB-Tree

In this section we present a brief performance evaluation of the integrated UB-Tree based on the GfK3D data set.

6.5.1 Benchmark environment

The measurements are conducted on a Sun Ultra 10 (Solaris 2.6) with one 440 MHz UltraSPARC-II processor and 512 MB of main memory. We use the TransBase HyperCube DBMS for our comparisons of the standard access methods of TransBase with the kernel integrated UB-Tree.

6.5.2 Data warehouse schema and queries

We use the hierarchical encoded data warehouse schema of GfK, i.e., the GfK3D instance (see Section 4.3). Figure 6.11 shows the DDL statement for the star schema. The foreign key relationship between the fact table and the dimension tables is implemented via the artificial keys generated by HC (Section 2.5).

In order to have a realistic scenario for our case study, we measure the following operations on the GfK DW:

- Reporting: processing of segmentation reports with following restrictions:
  - Product group series (PG): restriction to a two-month period, one country, and one product group (604 queries)
  - 4-Month period series (M4P): same as PG series, but just restricting a four-month period instead of a two-month period (12 queries)

```
CREATE TABLE TIME (
    TIME_CS INTEGER NULL,
    YEAR_ID NUMERIC NOT NULL,
    MONTH4_PERIOD_ID NUMERIC NOT NULL,
    MONTH2_PERIOD_ID NUMERIC NOT NULL
)
KEY IS YEAR_ID,MONTH4_PERIOD_ID,MONTH2_PERIOD_ID;

CREATE TABLE SEGMENT (
    SEGMENT_CS INTEGER NULL,
    COUNTRY_ID NUMERIC NOT NULL,
    REGION_ID NUMERIC NOT NULL,
    MICROMARKET_ID NUMERIC NOT NULL,
    OUTLET_KEY NUMERIC NOT NULL
)
KEY IS COUNTRY_ID,REGION_ID,MICROMARKET_ID,OUTLET_KEY;

CREATE TABLE PRODUCT(
    PRODUCT_CS INTEGER NULL,
    SECTOR_ID NUMERIC NOT NULL,
    CATEGORY_ID NUMERIC NOT NULL,
    PRODUCTGROUP_ID NUMERIC NOT NULL,
    ITEM_ID NUMERIC NOT NULL
)
KEY IS SECTOR_ID,CATEGORY_ID,PRODUCTGROUP_ID,ITEM_ID;

CREATE TABLE FACT (
    PRODUCT_CS INTEGER NOT NULL,
    SEGMENT_CS INTEGER NOT NULL,
    TIME_CS INTEGER NOT NULL,
    PD_PRICE INTEGER NOT NULL,
    PD_PACKAGE_PRICE INTEGER NOT NULL,
    PD_SALES INTEGER NOT NULL,
    PD_STOCK_OLD INTEGER NOT NULL,
    PD_STOCK_NEW INTEGER NOT NULL,
    PD_PURCHASE INTEGER NOT NULL,
    PD_TURNOVER INTEGER NOT NULL,
    PD_PROJECTION_FACTOR INTEGER NOT NULL,
    PD_DISTRIBUTION_FACTOR INTEGER NOT NULL,
    PD_UNIT_FACTOR INTEGER NOT NULL
)
HCKEY IS PRODUCT_CS, SEGMENT_CS, TIME_CS;
```

Figure 6.11: Create Statements for DW Schema

  - Category series (CAT): restriction to a two-month period, one country, and one category (30 queries)
  - All products (ALL): restriction to a two-month period, no restriction in the Product dimension but looping over different countries in the Segment dimension (16 queries)

- Maintenance: deletion and insertion of the data for a complete time period

Figure 6.12 shows a typical example query on the DW schema.

```
SELECT sum(PD_SALES)
FROM FACT, TIME, SEGMENT, PRODUCT
WHERE TIME.MONTH4_PERIOD_ID = 199801120 AND
SEGMENT.REGION_ID = 3203 AND
PRODUCT.SECTOR_ID = 162 AND
FACT.TIME_CS = TIME.TIME_CS AND
FACT.SEGMENT_CS = SEGMENT.SEGMENT_CS AND
FACT.PRODUCT_CS = PRODUCT.PRODUCT_CS;
```

Figure 6.12: Example Query

6.5.3 Compared access methods

We compare the following different access methods:

- PTS: a composite B-Tree with the key order (Product, Time, Segment)
- TPS: a composite B-Tree with the key order (Time, Product, Segment)
- UB: a UB-Tree on the attributes Product, Time, Segment
- MULT: a fact table indexed by three secondary indexes on the attributes Product, Time, Segment

Composite B-Trees require to choose one specific order of the index attributes resulting in a difficult key order decision problem. For the GfK fact table we investigate two different attribute orderings: the order (Time, Product, Segment) is investigated since all operations restrict the Time dimension and often the Product dimension. The order (Product, Time, Segment) is a promising alternative as the restriction the Product dimension is often very strong (the restriction to a product group is highly selective: in all cases below 2% with a median of 0,05%). In contrast to that, the UB-Tree does not require an attribute order as all dimensions are treated symmetrically. Due to its multidimensional nature, a UB-Tree on three dimensions Time, Product, Segment allows for utilizing the restrictions on all dimensions. The same holds for multiple secondary indexes on the three dimensions, but as secondary indexes are non-clustering, the materialization of the result tuples is expected to require more page accesses than for the UB-Tree.

Table 6.1: Index sizes in number of 2KB pages

	UB	**PTS**	**TPS**	**MULT**
Data pages	1863822	1510707	1513909	2041329
Index pages	20676	22499	22513	434308
Total	1884498	1533206	1536422	2475637

6.5.4 Index sizes and maintenance performance

Table 6.1 contains the index sizes for the fact table containing ≈ 43 million tuples.

There is no major difference in the number of data pages for PTS and TPS, whereas the UB-Tree is about 20% larger. This stems from the fact that the tuple compression on data pages for all TransBase tables is not yet done for the UB-Tree. MULT requires more data pages as it has to include extra space for tuple references needed for the secondary indexes. The index parts for UB, PTS, and TPS have almost the same size, whereas MULT requires more index pages, as there are three independent secondary indexes. With respect to the index size we can state that a UB-Tree with compression on the data pages does not require more space than a traditional composite B-Tree, but less than secondary indexes on the dimensions.

In order to evaluate the maintenance performance, we delete all fact data for a given two-month period and reinsert it into the fact table. We use the bulk-loading facility of the DBMS for these tasks. Note that the tuples to be inserted are sorted according to the Time dimension, as only the data of one time period is spooled in. Table 6.2 shows the execution times of maintenance operations for 2146779 tuples for the indexes.

Table 6.2: Time [in sec] for maintenance operations for 2146779 tuples

Operation	Time: UB	Time: PTS	Time: TPS	Time: MULT
Delete	148	945	161	1607
Bulk Insertion	561	785	336	2350

The results for the clustering indexes are not surprising: UB and TPS are able to identify the tuples to be deleted very fast, whereas PTS needs to scan through all data pages. With respect to loading, TPS has the benefit that one complete time interval is inserted; for UB and PTS the time period leads to multiple intervals in the key order and thus leads to higher loading costs. As expected for a secondary index, MULT takes much longer for the maintenance operations as more indexes have to be updated. This is a fundamental problem of secondary indexes - it is often suggested to first drop the indexes, delete the data, and then create the indexes again, if a large portion of the data has to be deleted or inserted.

6.5.5 Reporting performance

Regarding the reporting performance, our first observation is that the MULT can neither compete with the composite indexes nor with the UB-Tree for the reporting

series. Figure 6.13 shows the results of 20 PG queries, which demonstrate the poor performance of MULT; even for queries with small result sets it takes much more time than for the other indexes. As a consequence, we exclude the MULT from further measurements and take a closer look at the other indexes.

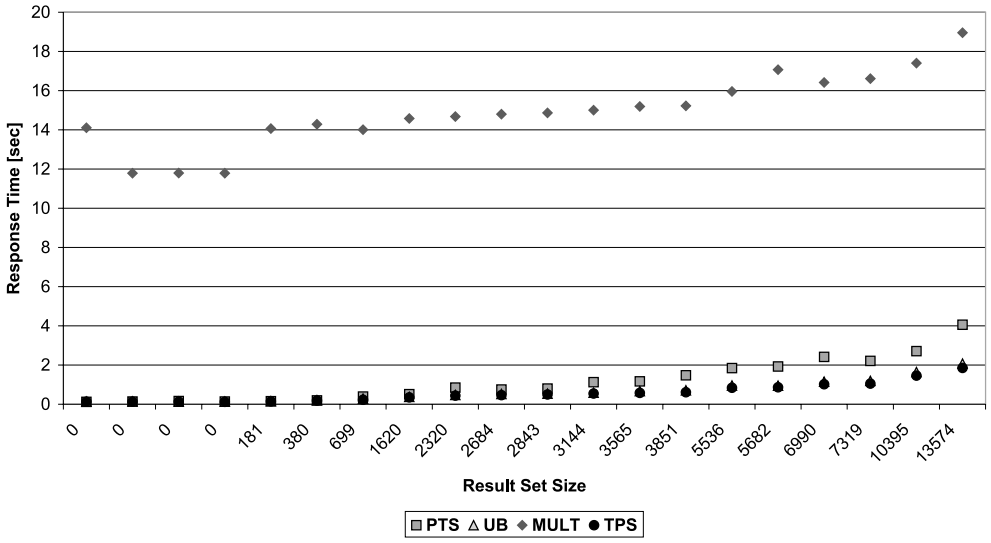


Figure 6.13: Response time [in sec] for the PG series with MULT

For the other indexes the complete PG results are given in Figure 6.14.

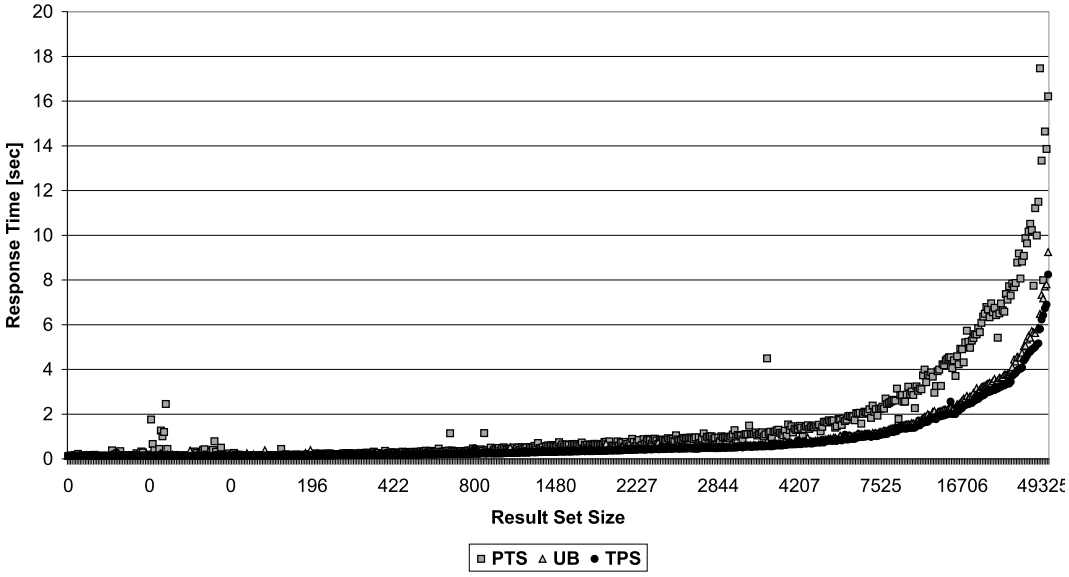


Figure 6.14: Response time [in sec] for the PG series

The composite B-Tree TPS is favored by point restriction in the first and the high selectivity on the second index attribute. In contrast, the composite PTS can only utilize the restriction on the Product dimension and, therefore, has to read

more pages. Even though the clustering according to the Z-curve of the UB-Tree is not optimal for this query set, the UB-Tree outperform PTS and is close to TPS, as it can utilize the restrictions in all dimensions.

When we relax the restriction on the Product dimension in the CAT series (i.e., a restriction only down to the category level), both composite B-Trees loose the advantage, as a larger and larger interval on the indexes has to be processed (see Figure 6.15).

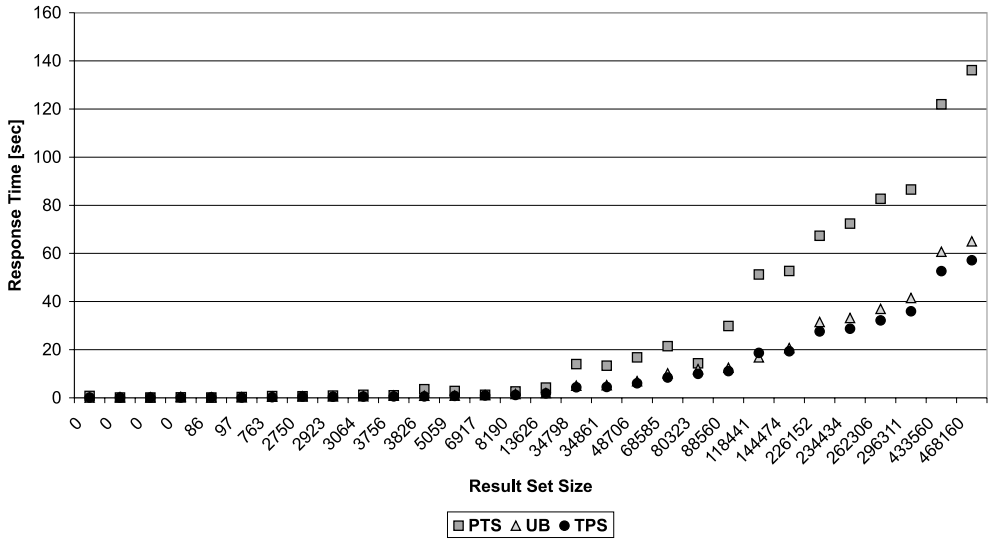


Figure 6.15: Response time for CAT series

For the ALL series, where the Product dimension is not restricted, the UB-Tree clearly outperforms TPS and PTS, as it can benefit more and more from the restrictions on the other dimensions (see Figure 6.16). PTS has to perform a full table scan, whereas TPS has to read all data for a complete two-month period.

The same holds for the M4P series shown in Figure 6.17. TPS no longer can benefit from the point restriction on the first attribute and thus can also not utilize the restriction in the Product dimension. PTS still has the strong restriction on the first attribute, but the UB-Tree uses all restrictions and thus shows the best performance.

The results of the reporting queries show that the UB-Tree outperforms the traditional index methods. It can compete with optimal clustering composite B-Trees and at the same time provides the flexibility for varying query patterns.

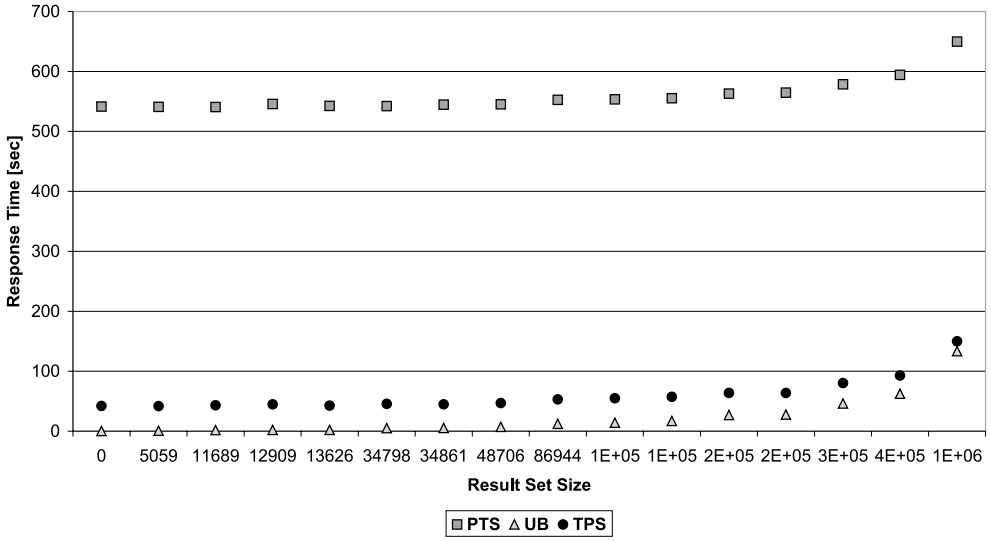


Figure 6.16: Response time for ALL series

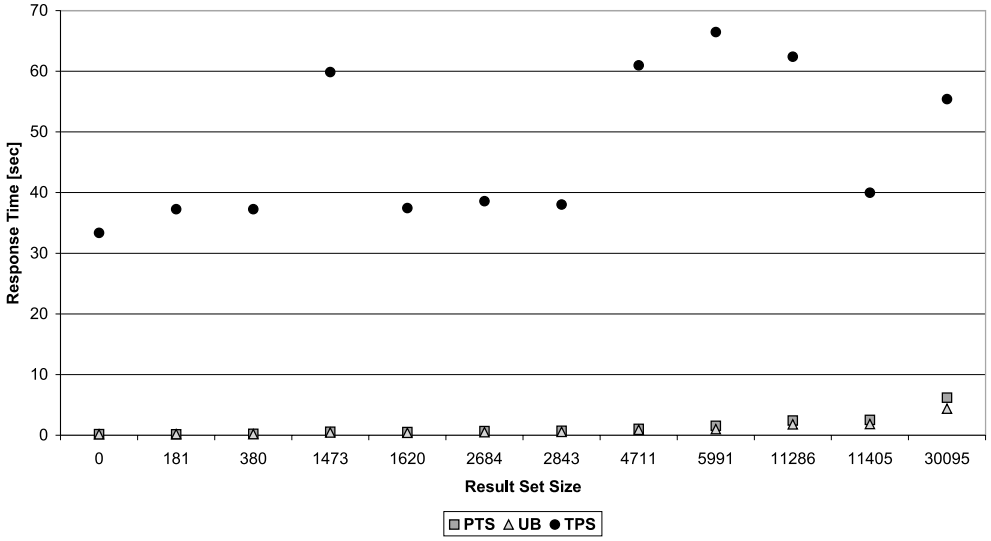


Figure 6.17: Response time for M4P series

6.6 Chapter notes and related work

Integration - is it worth the effort?

Multidimensional access methods are not widely supported by commercial database management systems despite their performance impacts in various application domains. This is mostly due to the fact that a kernel integration of these sophisticated data structures is considered to be a very costly and complex task. With the integration project we have shown that this is not the case for the UB-Tree, as it heavily relies on the well-known B-Tree, reducing the complexity of the additional algorithms to a minimum.

The performance gains show that the integration is worth the effort: the kernel integrated UB-Tree provides significantly better performance than traditional access methods in various application domains. The big advantage of the kernel integration in comparison with other approaches is the tight coupling with the query optimizer. As we do not rely on any TransBase specific features for the UB-Tree algorithms, we expect the same integration effort for other database systems as well, as long as they provide clustering B-Trees on computed keys. Summarizing our experiences, UB-Trees smoothly integrate into the indexing engine and extend the B-Tree concept in order to handle multiple dimensions symmetrically. They are extremely useful for both clustering tables as well as covering secondary indexes (secondary indexes that contain all attributes required by a given query) to speed up multidimensional range queries.

Related work

Integrating new index methods into a database kernel is often regarded as a too costly task. On the other side, database vendors have recognized the need for more flexible, powerful indexing methods, often tailored to specific application domains. As consequence, most vendors have extended their standard B-Trees and provide interfaces that allow the users to include their own functions for the index key computation. Some systems even allow the user to implement their own index structures in external modules. We will now point out the deficiencies of these approaches.

Function-based B-Trees For standard B-Trees the key of the index consists of a subset of the attributes of the underlying table. A more general idea is to use a function to compute the key values for the tuples. We will refer to this type of B-Trees as function-based B-Trees or short BF-Trees. For example, a standard B-Tree is a special instance of the BF-Tree where F is the projection of the key attributes from the tuple. B-Trees storing SOUNDEX codes or case-insensitive keys are other well-known examples. BF-Trees were motivated by the need to support indexing on user-defined types in object-relational systems. Commercial implementations are provided for example by function-based indexes in Oracle8i [ORA99], the high level indexing framework of IBM DB2 [CCF+99], or as indexes on computed columns in

MS SQL Server 2000 [MS000]. However, BF-Trees do not allow for the integration of new query algorithms, like the UB-Tree range query algorithm. Therefore, implementing the UB-Tree as a BF-Tree with the Z-value as function will not lead to the expected performance.

Extended index interfaces Some commercial database management systems provide even more enhanced indexing interfaces, which allow for implementation of arbitrary index structures by the user in external modules (e.g., Extensible Indexing API by Oracle [ORA99], Informix Datablade API [Inf99]). Analogous to the GiST framework (see next paragraph), the user has to provide a set of functions/operators that are used by the database server to access the index. The index itself can be either stored inside the database (e.g., as an IOT in Oracle) or in external files. The problem of these index interfaces is threefold: performance of the index, optimizer support, and locking and recovery. The performance problem of extended index interfaces has two aspects: first, only non-clustered indexes are supported. Index structures, whose performance is achieved by appropriate clustering, like the UB-Tree that clusters according to multiple dimensions, can therefore not be implemented via these interfaces. In addition, as the DBMS internal modules cannot be used, efficient page and tuple handling has to be implemented. This leads to significant coding effort for the index implementation. The coupling of the external index with the query optimizer is achieved by providing cost functions for the index operations. However, to our knowledge, there is no way to add rules to guide the optimizer with heuristics, which is very important to achieve optimal query plans. Another significant drawback of these 'add-on' approaches is the handling of locking and recovery. The external indexes are not tightly coupled with the DMBS locking and recovery services. As consequence, the index implementation has to take care of recovery issues itself [BSSJ99], and the lock granularity is often the complete index itself. Taking all these aspects into account, in case of the UB-Tree the kernel integration is much more favorable than an implementation as an external index.

The General Search Tree The General Search Tree(GiST) approach [HNP95] provides a single framework for any tree-based index structure. The GiST framework provides the basic functionality for trees, e.g., insertion, deletion, splitting, search, etc. The individual semantics of the index are provided by the user with a key class, which implements six key functions the basic functions rely on. As consequence, the user has only to change a small part of the code to implement various index methods (e.g., B-Trees, R-Trees). In general, the UB-Tree fits perfectly into the GiST framework, but efficient implementation would require more user control for the search algorithm and page splitting. The major drawback of the original GiST approach is the fixed query functionality - the user cannot adapt the search algorithm to the specific indexing technique, which in many application scenarios will lead to significant performance problems. The extension of [Aok98b] gives the user the control of the tree traversal during search and should suffice for an efficient range query implementation. Putting all together, with the extended GiST framework the benefits described in [Kor99] apply also for a UB-Tree implementation.

Chapter 7

Complete linearization of query boxes

As described in Section 6.2, a multidimensional query box decomposes into a set of intervals on the Z-curve. Instead of computing all of these intervals, the range query (**RQ**) algorithm works on the granularity of Z-regions. However, for some purposes it would be quite handy to have the complete linearization of the query box. One example is to optimize the RQ algorithm by saving post-filtering of pages that are completely contained in the query box. This becomes more frequent the larger the query boxes get, like the one depicted in Figure 7.1.

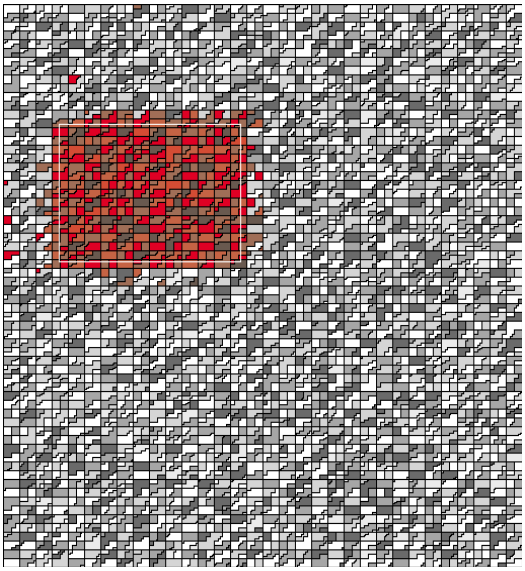


Figure 7.1: Visualization of a large query box

For computing the end of a Z-interval (the beginning is calculated by the NJI-algorithm), one has to find the last point inside the interval for a given point inside the interval. We call this end point the *next-jump-out* as from this point one is jumping out of the query box if one follows the Z-curve further. Formally, the problem can be specified as follows.

Definition 7.1 (Next-Jump-Out)
Let Q be the query box and cur be the Z-value of a point in the box, i.e., $Z^{-1}(cur) \in Q$. The 'next-jump-out' njo is the Z-value of the end point of the interval inside Q starting with cur, i.e.,

$$njo = NJO(cur) = \min\{y \in \mathcal{Z} | y \geq cur \wedge Z^{-1}(y) \in Q \ \wedge Z^{-1}(y+1) \notin Q\}.$$

7.1 The next-jump-out algorithm (NJO)

In the following we assume that the input to the next-jump-out algorithm is a point inside the query box[1]. Such, the first step of the algorithm is to identify bits in the current Z-value, which on change cause either that one dimension falls below the minimum of the query box or that one dimension exceeds the maximum of the query box. Then, in a second step, the Z-address is manipulated to fulfill the condition for the next-jump-out. In the next subsection we describe the details of the algorithm and provide a proof of correctness for the algorithm.

7.1.1 The NJO Algorithm

A naive solution to the NJO problem would be to start with the current Z-value *cur* and increment it by one until the resulting Z-address is outside the query box Q; the next-jump-out is then the current value minus one. This algorithm is definitely correct but is very inefficient as it requires n increments and n tests if the Z-address is in Q for a Z-interval of length n. In the following we present an algorithm for NJO, which has complexity $O(l)$ for Z-addresses of length l. The basic idea is to find the smallest Z-address that is larger than the current address and which is outside Q. Decrementing this value by 1 leads us to the real NJO.

We first introduce the notion of *violations*. To this end, we have to enhance our notation for bit strings. Given a bit b, $\bar{b}$ denotes the negated value of b, i.e., $\bar{b} = \neg\, b$

Definition 7.2 (Violation)
Given a Z-address z and a query box Q with $Z^{-1}(z) \in Q$. A violation of z with respect to Q is a bit position i, $0 \leq i \leq |z|$ that causes z to violate Q, i.e., $Z^{-1}(z) \notin Q$ if the bit at position i, and possibly bits with positions smaller than i, is changed. Formally, for $z = z_{l-1}z_{l-2}\ldots z_0$ position i is a violation, iff $Z^{-1}(z') = Z^{-1}(z_{l-1}\ldots z_{i+1}\bar{z}_i z'_{i-1}\ldots z'_0) \notin Q$, with $z'_j \in \{0,1\}, 0 \leq j \leq i-1$.

We distinguish two kinds of violations: *MIN-violations* and *MAX-violations*. MIN-violations cause the current Z-address to fall below the minimum of the query box in one dimension, whereas MAX-violations cause to exceed the maximum of the query box in one dimension. We call the violation with the lowest bit position of a Z-value z, i.e., the bit position with the lowest bit value, the *smallest violation* of z with respect to a query box Q.

[1] If the input point is not in the query box, the NJI algorithm has to be called before the NJO algorithm to guarantee a correct input.

Example 7.1: Violations

In the following, we will use a two-dimensional, 8×8 universe for our illustrations of the algorithm, i.e., $\Omega = \mathbb{D} \times \mathbb{D}$, with $\mathbb{D} = \{x | 0 \leq x \leq 7\}$.

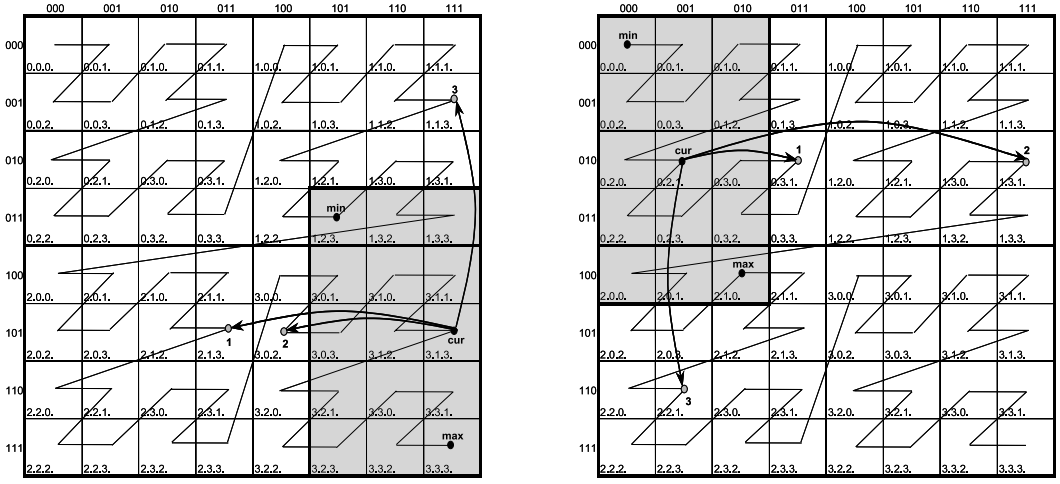
Given the query box $Q = [[min, max]]$ with $min = 001001$ and $max = 101100$ and the current Z-address $cur = 001100$. We show which addresses cur' result from the various violations.

For the given configuration we have a MIN-violation at position 3, because if we set $cur_3 = 0$, then $cur'^2 = 000 < min^2 = 010$. Another MIN-violation can be found at bit position 2, as setting $cur_2 = 0$ leads to $cur'^1 = 000 < min^1 = 001$.

A MAX-violation is located at position 5, as with $cur_5 = 1$ and $cur_1 = 1$, $cur''^2 = 111 > max^2 = 110$. And finally, we have a MAX-violation at bit position 0. Setting $cur_0 = 1$ achieves $cur''^1 = 011 > max^1 = 010$.

This example shows that it is sometimes necessary to change more than two bits to achieve a violation.

Figure 7.2 gives a graphical illustration of MIN-violations and MAX-violations for two other query boxes.



(a) MIN-violations (b) MAX-violations

Figure 7.2: Violations of a query box

◇

7.1.1.1 Finding violations

For determining violations, it suffices to treat all dimensions individually, as only bits of a Z-value corresponding to the same dimension may cause a violation of the query box in this dimension.

To find all violations it is necessary to completely scan through the current address *cur* and the two addresses *min* and *max* that specify the query box. In the algorithm we scan through the addresses bit per bit beginning with the bit at the highest position by two for-loops. The first loop increments the dimension while the second one loops over all bits in the dimension. For every bit of the address the algorithm compares now the current bit from *cur* with the current bits from *min* and *max*. If it is possible to achieve a violation by inverting the current bit then this position will be marked. Figure 7.3 shows the pseudo-algorithm for finding all possible violations. For the following manipulation algorithm we require the violations to be returned in ascending order. For the algorithms we use a mapping between a bit of the binary representation of a dimension value and the corresponding bit in the Z-address. The function `bp(i,j)` returns the bit position p in the Z-value of bit j of dimension i; `dim(p)` returns the dimension d bit position p belongs to. `maxpos(i)` denotes the maximum bit position for dimension i. In the following, we represent bit positions as integers. For the algorithm we further require one list to store the found violations as well as two variables to store violation candidates. Violation candidates are necessary to identify violations that require the change of more than one bit in the current Z-value. We use the following intuitive operations on a list `L` of type `LIST`:

- `append(L,a)` appends an element `a` at the end of the list
- `isempty(L)` checks whether the list is empty or not
- `pop(L)` returns the first element in the list and deletes it
- `sort(L)` sorts the list in ascending order

For illustration purposes, the presented pseudo-code for finding violations loops over the dimensions sequentially. This causes multiple passes over the Z-addresses. In a real implementation, it suffices to pass through the Z-addresses once, looking at all dimensions simultaneously.

```
LIST FindViolations(Z-value min, Z-value max, Z-value cur) {

   LIST violations={};
   int min_cand, max_cand;
   int p,i,j;
   int curbit, minbit, maxbit;
   bool minfinished, maxfinished;

   //for each dimension check for violations
   for(i=1; i<=d;i++) {
      //initialize lists
      min_cand=-1; max_cand=-1;
      minfinished=false; maxfinished=false;

      //loop over all bits of the dimension, beginning with the highest valued one
      for(j=maxpos(i); j>=0; j--) {
         p=bp(i,j);
         curbit=cur_p;
         minbit=min_p;
         maxbit=max_p;
         //looking for MIN violations
         if(!minfinished) {
            if(curbit==minbit && minbit==1 && min_cand==-1)
               //found MIN-violation in p
               append(violations,p);
            else if(curbit==1 && minbit==0 && min_cand==-1)
               //found a MIN-violation candidate
               min_cand=p;
            else if(minbit==1 && min_cand >= 0)
            {  //the candidate is really a violation
               append(violations,min_cand);
               minfinished=true;
            }
         }
         //looking for MAX violations
         if(!maxfinished) {
            if(curbit==maxbit && maxbit==0 && max_cand==-1)
               //found a MAX-violation in p
               append(violations,p);
            else if(curbit==0 && maxbit==1 && max_cand==-1)
               //found a MAX-violation candidate
               max_cand=p;
            else if(maxbit==0 && max_cand>=0)
            {  //the candidate is really a violation
               append(violations,max_cand);
               maxfinished=true;
            }
         }
      }
   }
   sort(violations);
   return violations;
}
```

Figure 7.3: Finding Violations (FV) Algorithm

7.1.1.2 Manipulating current Z-value based on violations

With the algorithm described in the previous section we identify bits that cause a violation of the query box if they are changed. However, we still have to make sure that the other conditions for the NJO hold, namely that the new address is larger than the current address and that there is no smaller address which is also outside the query box.

First, we consider the special case in which no violation for the Z-address cur with respect to query box $Q = [[min, max]]$ was found. If for each dimension one can not get below the minimum or can not exceed the maximum then Q covers the complete universe, i.e., min and max correspond to the minimal resp. maximal Z-address. Consequently, the next-jump-out njo is the maximal Z-address, i.e., $njo = max$.

Now let us consider the cases where we have found violations in cur. We are starting with the smallest violation found for cur. The manipulation of cur depends on the type of the violation. We first look at the case of a MAX-violation because using this violation already leads to a Z-value $cur' > cur$.

MAX-violation

Let us assume a MAX-violation v in dimension i of Z-address cur to be the smallest violation with respect to query box $Q = [[min, max]]$. The next-jump-out njo is obtained by decrementing the Z-value cur' by one, which is gained by manipulating cur as follows. First set the dimension i to the value $max^i + 1$ thus causing the violation of Q. To get the smallest Z-value set all bits of other dimensions at positions smaller than v to zero. Consequently, $njo = cur' - 1$, with

$$\begin{gathered} cur'^i = max^i + 1 \\ cur'^k_j = cur^k_j \ \forall j, k \ with \ k \neq i \wedge j > v \\ cur'^k_j = 0 \ \forall j, k \ with \ k \neq i \wedge j < v \end{gathered}$$

Example 7.2: MAX-violation

Let $min = 0.1.0 = 000100$, $max = 3.0.3 = 110011$, and $cur = 0.3.3 = 001111$ (cf. Figure 7.4). There is a max-violation in dimension 1 at position 4. Setting the bits of dimension 1 to $max^1 + 1$ leads to $cur' = 011110 = 1.3.2$. Setting the bits of dimension 2 at positions smaller than 4 to 0 results in $cur'' = 010100 = 1.1.0$; decrementing cur'' by 1 gets the correct $njo = 010011 = 1.0.3$.

◇

MIN-violation

In case that the smallest violation v is a MIN-violation the manipulation of the Z-address cur gets a bit more complex. The problem is that if we set $cur_v = 0$ in

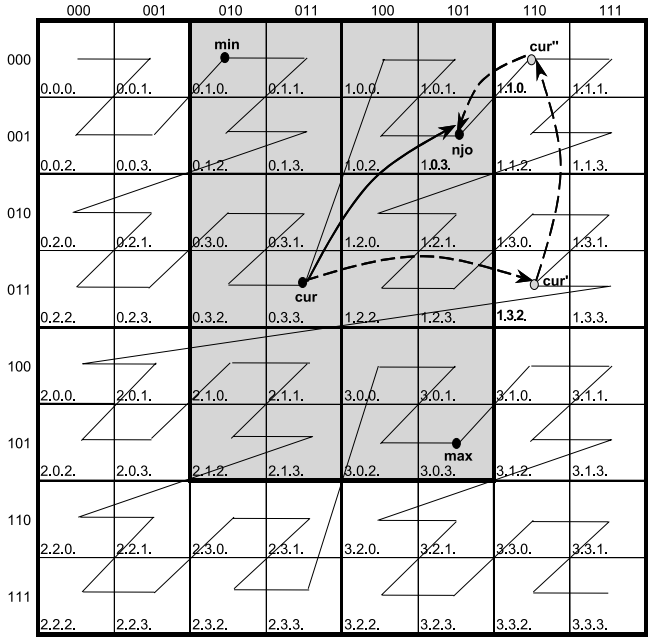


Figure 7.4: MAX-violation

order to achieve the violation then the generated Z-value is smaller than *cur*. Thus, we have to find another bit position p, whose value can be set to 1 and which is larger than v. Furthermore, we have to check if p is from the same dimension as v. Because the MIN-violation disappears if we set a bit of the same dimension at a higher position. Consequently, there are two conditions to achieve a MIN-violation: first, there is a larger position in *cur* with value 0, so that it can be set, and second, the dimension of this position is different to the dimension in which the smallest violation was found.

For the algorithm we therefore have to check four constellations.

Case 1: In this case, there exists a bit position *pos* in *cur* that has the value 0 and does not belong to the same dimension as v, i.e., $\exists pos : pos > v \wedge cur_{pos} = 0 \wedge dim(v) \neq dim(pos)$. The next-jump-out njo is then achieved by setting the bit at position *pos* to 1 and setting all bits at smaller positions to 0 and finally decrementing this value by 1. Formally, let $pos = min\{p | p > v \wedge cur_p = 0\}$, if

$$dim(v) \neq dim(pos)$$

$$\Downarrow$$

$$njo = cur' - 1, \quad \text{with} \quad cur'_j = \begin{cases} cur_j, & \text{if} \quad j > pos \\ 1, & \text{if} \quad j = pos \\ 0, & \text{if} \quad j < pos \end{cases}$$

Note: obtaining njo in this case can be optimized by just setting all bits at positions smaller than *pos* to 1 (*pos* has already the value 0).

Example 7.3: MIN-violation: Constellation 1

Let $min = 0.2.3 = 001011$, $max = 3.0.2 = 110010$, and $cur = 0.3.2 = 001110$ (cf. Figure 7.5). cur has a min-violation at position 1, i.e., in dimension 2. As the next zero bit of cur at position 4 belongs to dimension 1, we can safely set that bit resulting in $cur' = 011110 = 1.3.2$. We then set all bits at positions smaller than 4 to 0 getting $cur'' = 010000 = 1.0.0$ and with the final decrement leading to $njo = 001111 = 0.3.3$.

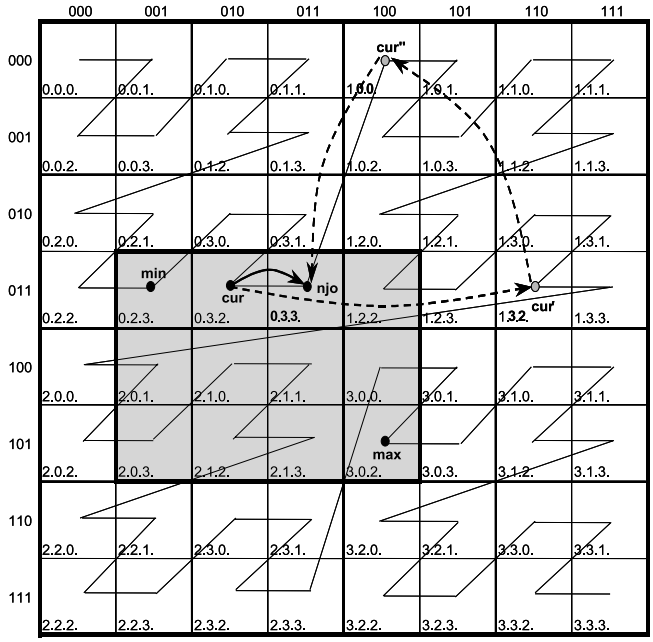


Figure 7.5: MIN-violation: Constellation 1

◇

Case 2: In this constellation, there is also a bit position pos which is larger than v and has the value 0, but this time it corresponds to the same dimension as v, i.e., $\exists pos : pos > v \wedge cur_{pos} = 0 \wedge dim(v) = dim(pos)$. Setting this bit will cause the MIN-violation v to be ineffective. However, if there is another MIN-violation w with $w > v \wedge dim(w) \neq dim(pos)$, we can use this violation instead of v and handle it as described in the first case. Formally, the second condition for $pos = min\{p | p > v \wedge cur_p = 0\}$ to be used for the manipulation is

$$\exists w, v < w < pos \text{ with } w \text{ is MIN-violation} \wedge dim(w) \neq dim(pos)$$

$$\Downarrow$$

$$njo_j = \begin{cases} cur_j, & \text{if } j > pos \\ 0, & \text{if } j = pos \\ 1, & \text{if } j < pos \end{cases}$$

Example 7.4: MIN-violation: Constellation 2

Let $min = cur = 0.3.0 = 001100$ and $max = 3.0.2 = 110010$ (cf. Figure 7.6). The min-violation of *cur* exists at position 2 belonging to dimension 1. As the next zero bit at position 4 also belongs to dimension 1 this bit can not be set to 1 without further checking as this may cause the min-violation to disappear. As there is also a min-violation in dimension 2 at position 3, the situation corresponds to the one in the first case. Consequently, the $njo = 001111 = 0.3.3$ with $cur' = 011100 = 1.3.0$ and $cur'' = 010000 = 1.0.0$ as intermediate steps.

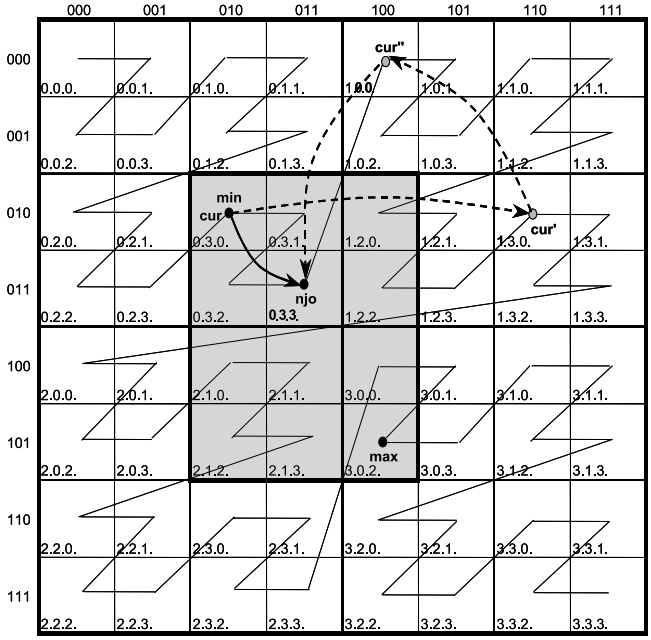


Figure 7.6: MIN-violation: Constellation 2

◇

Case 3: If cases 1 and 2 do not apply, i.e., *pos* is from another dimension than v and there is no other MIN-violation in between, then *pos* cannot be used for the manipulation. Unless *pos* is at the same time a MAX-violation, which we then can use directly. So, with $pos = min\{p|p > v \wedge cur_p = 0\}$, if

$$dim(pos) = dim(v) \wedge$$
$$\nexists w, v < w < pos \text{ with } w \text{ is MIN-violation} \wedge$$
$$pos \text{ is MAX-violation}$$

$$\Downarrow$$

$$njo = cur' - 1, \text{ with}$$
$$cur'^i = max^i + 1, i = dim(pos)$$
$$cur'^k_j = cur^k_j \ \forall j, k \text{ with } k \neq i \wedge j > pos$$
$$cur'^k_j = 0 \ \forall j, k \text{ with } k \neq i \wedge j < pos$$

Example 7.5: MIN-violation: Constellation 3

Let $min = 0.2.1 = 001001$, $max = 2.1.2 = 100110$, and $cur = 0.2.3 = 001011$ (cf. Figure 7.7). There is a MIN-violation of dimension 1 at position 0, but the next zero bit at position 2 also belongs to dimension 1 and there is no MIN-violation in dimension 2. Still, dimension 1 has a MAX-violation at this position that we process as follows: we set $cur^1 = max^1 + 1$ ($cur' = 001111 = 0.3.3$) and all bits of dimension 2 at positions smaller than 2 to zero ($cur'' = 001101 = 0.3.1$); finally decrementing this value by one achieves $njo = 001100 = 0.3.0$.

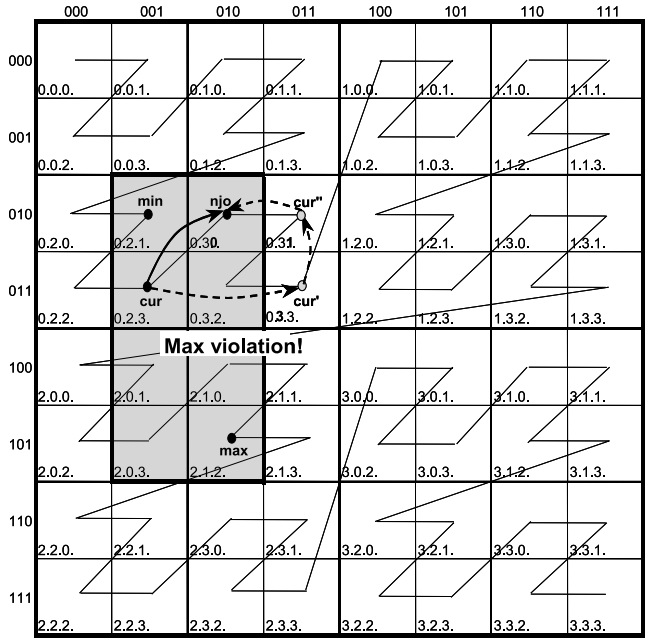


Figure 7.7: MIN-violation: Constellation 3

◇

Case 4: If none of the cases 1-3 applies for position pos, we continue with the next larger position that has the value 0. However, we might end up with finding no bit with value 0, which corresponds to a MAX-violation or which we safely can set for a MIN-violation. In this situation, the NJO corresponds to the maximal Z-address, as we are not able to find a manipulation in cur that violates Q and at the same time generates a Z-address larger than cur.

$$\forall pos, pos > v \wedge cur_{pos} = 0 : \ dim(pos) = dim(v) \wedge$$
$$pos \neq \text{MAX-violation} \wedge$$
$$\nexists w, v < w < pos : w \text{ is MIN-violation}$$

$$\Downarrow$$

$$njo = max$$

Example 7.6: MIN-violation: Constellation 4

Let $min = 1.2.3 = 011011$, $max = 3.3.3 = 111111$, and $cur = 3.2.3 = 111011$ (cf. Figure 7.8) . For this constellation we have min-violations of dimension 1 at positions 0 and 4 and a min-violation of dimension 2 at position 5. Starting with the violation at position 0 the next bit which can be set is at position 2. As it belongs also to dimension 1 and there is no min-violation of the other dimension in-between and there are no max-violations in this case, this bit cannot be set. As it is the only zero-bit in *cur*, we are not able to find a bit to set. Hence, the next-jump-out has to be the maximum address.

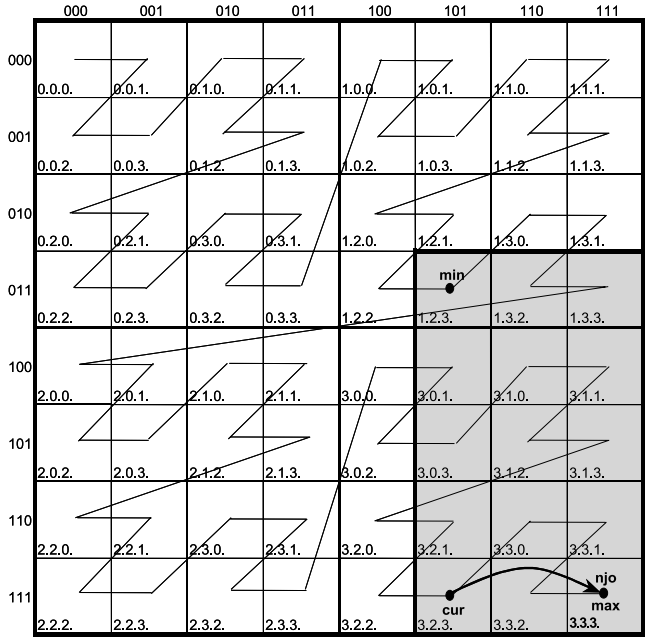


Figure 7.8: MIN-violation: Constellation 4

◇

Figure 7.9 shows the complete algorithm `getNJO` for manipulating the current Z-address in pseudo-code. Besides the query box specification `min` and `max` and the current Z-value `cur`, the algorithm uses the output of the FV-algorithm, the list of `violations` sorted in ascending order, as input. As described above, the algorithm starts with the smallest violation and passes backward through the Z-address to find the bits to be manipulated.

```
Z-value getNJO(Z-value min, Z-value max, Z-value cur, LIST violations) {

//Assumption:  list of violations is sorted in order of ascending bit positions

   Z-value njo, tmp;
   Integer i,j,dimv,v,c;
   Boolean vbetween=FALSE;

   if(isempty(violations))
      //the query box covers the complete universe
      return max;
   else {
      while(!isempty(violations)) {
         v=pop(violations);
         dimv=dim(v);
         if(cur[v]==0) { //v is MAX-violation
            tmp^dim = max^dim + 1;
            for(i=MAXPOS;i>=0;i--) {
               if(i > v && dim(i) != dimv)
                  tmp[i]=cur[i];
               if(i < v && dim(i) != dimv)
                  tmp[i]=0;
            }
            return (tmp-1);
         }
         else { // v is MIN-violation
            if(!isempty(violations)) c=pop(violations);
            else c=v;
            for(i=v+1;i <= MAXPOS;i++) {
               if(cur[i] == 0) {
                  if(dimv != dim(i) || vbetween) {
                     for(j=MAXPOS;j >= 0;j--) {
                        if(j >= i) njo[j]=cur[j];
                        else njo[j]=1;
                     }
                     return njo;
                  }
                  //the zero-bit is also MAX-violation handle this case
                  else if(c == i) {
                     push(violations,i);
                     break;
                  }
               }
               else if(i == c && dimv != dim(i)) {
                  vbetween=TRUE;
                  if(!isempty(violations)) c=pop(violations);
               }
            }
         //if there no zero bit that can be safely set then NJO=max
         if(i>MAXPOS) return max;
         }
      }
   }
}
```

Figure 7.9: Manipulation algorithm of NJO

7.1.2 Proof of correctness

In this section, we provide the formal proof of the correctness of the above described algorithm.

Given a query box Q defined by the two Z-addresses min and max and a current address $cur \in Q$, we show that the resulting Z-value njo is really the next-jump-out of Q. According to Definition 7.1 we have to show the following:

1. $njo \in Q \wedge (njo + 1) \notin Q$
2. $\forall x, cur \leq x \leq njo : x \in Q$

It is equivalent to show

1. $njo' = (njo + 1) \notin Q$
2. $\forall x, cur \leq x < njo' : x \in Q$

We use this transformation of the goals of the proof, as the algorithm always generates njo' as an intermediate step.

The proof is based on the structure of cur, i.e., on the nature of the violations found in the current address. In the following, v denotes the smallest violation found in cur.

Assume there is an address α between cur and njo' that is not inside Q, i.e., $\exists \alpha \in \mathcal{Z}$ with $cur \leq \alpha < njo' \wedge \alpha \notin Q$. We will show that such an α can not exist.

Let m be the smallest violation in cur and z be the highest manipulated bit of cur leading to njo. Consequently, $\forall j > z : \ cur_j = njo'_j$.

$$\begin{aligned} \alpha \notin Q &\Rightarrow \exists i : \alpha^i > max^i \vee min^i < \alpha^i \\ &\Rightarrow \exists \text{ violation } v \text{ in } cur \text{ with } z \leq v \leq m \wedge dim(v) = i \\ cur \leq \alpha < njo' &\Rightarrow \forall j > z : \ \alpha_j = cur_j \\ &\Rightarrow v \leq z \end{aligned}$$

In the following, we examine the following four possibilities:

1. m is MAX-violation, v is MIN-violation
2. m is MAX-violation, v is MAX-violation
3. m is MIN-violation, v is MIN-violation
4. m is MIN-violation, v is MAX-violation

1. Case: m is MAX-violation, v is MIN-violation

$$\begin{aligned} m \text{ is MAX-violation} &\Rightarrow m = z \\ &\Rightarrow v = z \\ &\rightarrow \text{ contradiction to } m \text{ is MAX-violation and } v \text{ is MIN-violation} \end{aligned}$$

2. Case: m is MAX-violation, v is MAX-violation

$$\begin{array}{rcl} m \text{ is MAX-violation} & \Rightarrow & m = z = v \\ \forall j < z:\ njo'_j = 0 \wedge dim(j) \neq dim(z) & \wedge & \\ njo'_{dim(z)} = max_{dim(z)} + 1 & \Rightarrow & \alpha \geq njo' \\ & \rightarrow & \text{contradiction to } \alpha < njo' \end{array}$$

3. Case: m is MIN-violation, v is MIN-violation

$$\begin{array}{lcl} v \text{ is MIN-violation} & \Rightarrow & \exists \text{ bit } b : z \leq b < v \wedge cur_b = 0, \alpha_b = 1 \\ & & \wedge dim(b) \neq dim(v) \\ dim(b) \neq dim(m) & \overset{\text{MIN-Case1 (see Pg. 89)}}{\Rightarrow} & z = b \\ & \overset{\forall i<z:\, njo'_i=0}{\Rightarrow} & \alpha \geq njo' \\ & \rightarrow & \text{contradiction to } \alpha < njo' \\ dim(b) = dim(m) & \Rightarrow & dim(v) \neq dim(m) \\ & \overset{\text{MIN-Case 2 (see Pg. 90)}}{\Longrightarrow} & z = b \\ & \overset{\forall i<z:\, njo'_i=0}{\Rightarrow} & \alpha \geq njo' \\ & \rightarrow & \text{contradiction to } \alpha < njo' \end{array}$$

4. Case: m is MIN-violation, v is MAX-violation

In this case $m = z$, as:

$$\begin{array}{lcl} dim(m) \neq dim(v) & \overset{\text{MIN-Case 2 (see Pg. 90)}}{\Longrightarrow} & z = m \\ dim(m) = dim(v) & \overset{\text{MIN-Case 3 (see Pg. 91)}}{\Longrightarrow} & z = m \end{array}$$

$$\begin{array}{lcl} z = m \wedge dim(m) \neq dim(v) & \Rightarrow & njo'_z = 1 \wedge \forall j < z:\ njo'_j = 0 \\ & \Rightarrow & \alpha \geq njo' \\ & \rightarrow & \text{contradiction to } \alpha < njo' \\ z = m \wedge dim(m) = dim(v) & \Rightarrow & \alpha \geq njo' \text{ analogous to 2. Case} \\ & \rightarrow & \text{contradiction to } \alpha < njo' \end{array}$$

□

7.1.3 Performance analysis

For a brief comparison of the complexity of the NJO-algorithm, we concentrate on the number of required bit operations. We assume that reading and writing a bit, as well as, comparing two bits comes with the same cost, e.g., for comparing two bits we require three operations: reading both bits (two operations) and one operation for comparing them. Let l be the length of the Z-address. We show that the NJO-algorithm has a complexity of $O(l)$ bit operations.

Finding violations

As described in Section 7.1.1.1, for finding all violations we have to scan through the three addresses *min*, *max*, and *cur* causing $3 * l$ operations for reading all bits. In addition, we require $2 * l$ operations for comparing the bits of *min* and *cur* on the one hand, and *max* and *cur* on the other hand. Thus, we find all violations with $5 * l$ bit operations in worst case.

Manipulating the current address

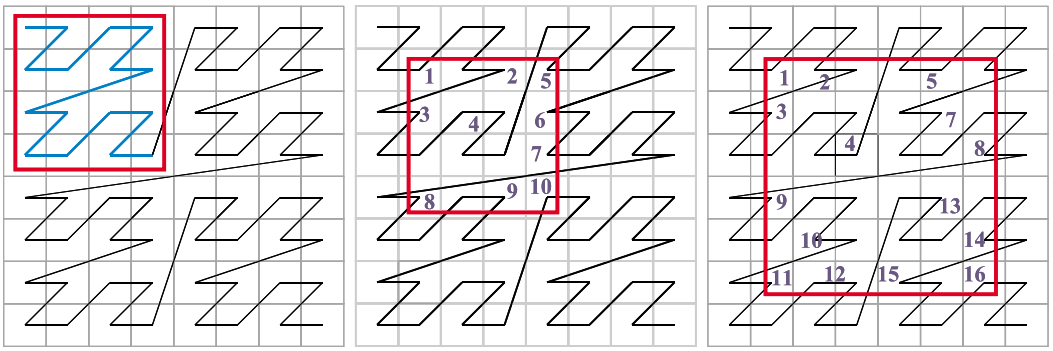
After finding the violations, we now have to look at the cost of manipulating the current Z-value in order to get the correct next-jump-out. For analyzing the cost for this step we consider the algorithm given in Figure 7.9. In the case of a MAX-violation we require a read and write operation for each bit in the Z-address. With a smart implementation this contains already the cost for the subsequent decrement. When we are handling a MIN-violation the situation is a bit more complex. First, we may have to read all bits before getting to the bit which is the starting point of the manipulation. Secondly, we either have to set all bits or we have to go to the MAX-violation case. In total, the manipulation of the address may cost up to $3 * l$ bit operations: starting with a MIN-violation that is finally handled by a MAX-violation.

7.2 Linearization of a query box

Given the NJO-algorithm, we can completely linearize a query box without looking at the data first. A query box Q defined by two tuples ql and qh with $Z(ql) \leq Z(qh)$ is composed by a set of Z-intervals $[nji_0, njo_0], \ldots, [nji_j, njo_j], \ldots, [nji_n, njo_n]$, where $nji_0 = Z(ql),\ njo_j = NJO(nji_j)$, and $nji_{j+1} = NJI(njo_j)$.

The problem of the linearization of query boxes is the huge number of intervals possibly generated. The number of intervals may vary between 1 and $\frac{|\Omega|}{2}$[2]. Figure 7.10 shows examples of query boxes and their linearization. The number of intervals does not only depend on the size of the query box but rather on the location of the box as Figure 7.10(a) and Figure 7.10(b) illustrate.

[2] $\frac{|\Omega|}{2}$ is the theoretical upper bound given by the maximal number of partitions of $\mathcal{Z}$ into intervals: the worst case is achieved assuming intervals of length 1, every second Z-value being part of the query box.



(a) Best case: 1 Z-interval for query box

(b) Same query box: 10 intervals due to different position

(c) Bad case: 16 Z-intervals

Figure 7.10: Linearization of query boxes

7.3 Reducing post filtering

This section presents a practical application of the NJO-algorithm leading to improved range query processing w.r. to CPU cost. For each Z-region/page retrieved by the range query algorithm, all tuples on the page have to be post-filtered with the search predicate. If the query predicate contains only index attributes, post-filtering can be saved for pages completely contained in the query box and therefore the whole range query processing can be optimized. If the query also restricts non-index attributes at least the post-filtering w.r. to the index attributes can be saved.

The idea is to compute for each next-jump-in nji also the next-jump-out njo. With this information and the address of the next Z-region, given by the following page separator sep, we can decide upon the post-filtering of pages. Figure 7.11 depicts the three constellations of Z-regions/pages and intervals of the query box.

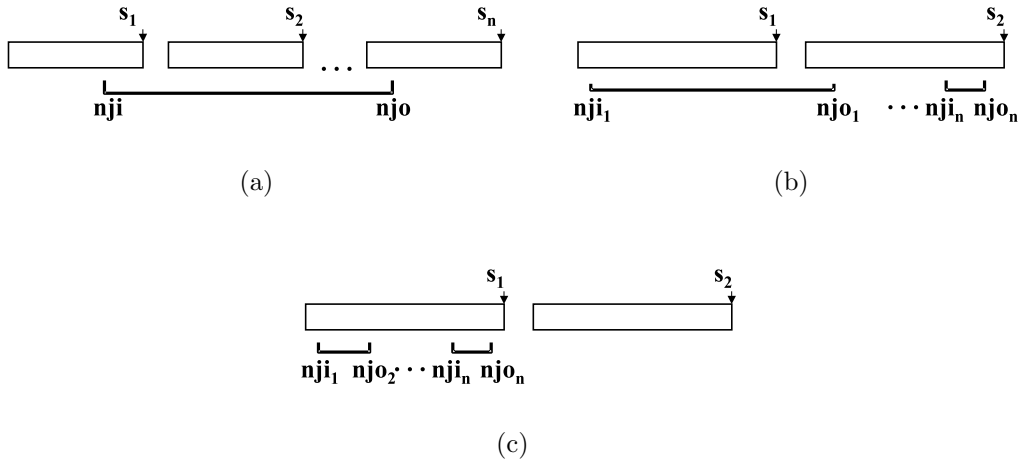


Figure 7.11: Optimizing post-filtering

Figure 7.11(a) illustrates the situation in which post-filtering can be reduced: s_1 denotes the Z-region containing the beginning of the current Z-interval $[nji, njo]$. The separator s_2 of the following page is smaller than the end of the interval njo. This means that all tuples on the page identified by s_2 are inside the query box and the post-filtering of those tuples can be omitted. Skipping the post-filtering is continued for all subsequent pages with $s_i \leq njo$. For the first page with $s_n > njo$ post-filtering has to be applied again. This corresponds to the case shown in Figure 7.11(b), where $njo < s_2$ and therefore requiring page s_2 to be post-filtered. If njo is even smaller than the separator s_1 of the current page (see Figure 7.11(c)) then the RQ continues like in the original version. Such, in all cases, the range query algorithm continues with computing the next-jump-in with the separator of the current page, i.e., the pages retrieved last. Figure 7.12 shows the pseudo-code for the optimized range query algorithm. `getNextRegionSeparator(cur)` retrieves the next region separator larger than `cur` from the index part. As usually the current path through the index is cached, this operation does not cause additional I/O.

```
rangeQuery(Tuple ql, Tuple qh) {
   Z-value start = Z(ql);
   Z-value cur = start;
   Z-value njo = getNextJumpOut(cur,start,end);
   Z-value end = Z(qh);
   Z-value sep;
   Page page = { };
   while (1)
   {
      cur = getRegionSeparator(cur);
      page = getPage(cur);
      postFilterPage(page, ql, qh);
      sep = getNextRegionSeparator(cur);
      while(sep <= njo) // next page is completely inside the query box
      {
         page = getPage(sep);
         output(page);
         sep = getNextRegionSeparator(sep);
      }
      if(njo > cur) // next page is partly inside the query box
      {
         page=getPage(sep);
         postFilterPage(page, ql, qh);
         cur = sep;
      }
      if ( cur >= end ) break; // termination:  query box completely covered
      // compute next-jump-in
      cur = getNextJumpIn(cur, start, end);
      njo = getNextJumpOut(cur, start, end);
   }
}
```

Figure 7.12: Optimized RQ algorithm

The optimized RQ algorithm saves significant CPU cost for the range query

processing by reducing the number of tuples that need to be post-filtered. In worst case, i.e., if no Z-regions are completely contained in the query box, the optimized version requires additional CPU for computing the njo for each nji.

7.4 Chapter notes

In this chapter we have introduced the 'inverse' function to the NJI-algorithm, the next-jump-out (NJO) algorithm computing the last Z-value of a Z-interval of a query box given any point inside the interval. This allows for computing all Z-intervals of a query box without looking at the data first. Describing a query in form of a list of Z-intervals is very interesting for various issues of query optimization, e.g., cardinality estimation. The major problem with the complete linearization of query boxes is the very large number of Z-intervals possibly created. The number of intervals does not only depend on the number of dimensions or the size of the query box but more crucially on the location of the query box. This large set of intervals causes on the one hand a large storage overhead and on the other hand leads to an increase in the CPU cost. Consequently, the complete linearization is only practical if the number of intervals is bounded (e.g., by enlarging the query box).

We have sketched one important application of the NJO algorithm, namely the optimization of the RQ-algorithm by reducing post-filtering of pages. For this application, the additional CPU cost is bound by the number of loaded pages and the expected savings of CPU cost are significant.

Related work

To the best of our knowledge, this is the first and only discussion of the NJO-algorithm, i.e., the complete linearization of a query box.

Chapter 8

Approximation of Z-regions and its applications

In Section 6.1 we introduce the Z-regions as a key concept of the UB-Tree query algorithms. Z-regions allow for having two views on the same thing: a Z-region is on the one side a one-dimensional interval on the Z-curve and on the other side represents a sub-space of the multidimensional universe covered by the Z-curve in the specified interval. Even though it is easy to specify a Z-region as an interval on the Z-curve, it is difficult to specify the exact extent of the Z-region in multidimensional space, due to the nature of the Z-curve. For some problems, it would be beneficial to have an easier description of the spatial extent of a Z-region, e.g., for computing the distance of a point to a Z-region or the border of a Z-region.

In this section we introduce how to approximate a Z-region by a minimum bounding box (MBB) and show how such an approach can be used for various advanced applications on UB-Trees, especially in cases where a distance to a Z-region has to be computed.

8.1 Computing the MBB for a Z-region

The spatial extent of Z-regions is defined by the image of the Z-curve in the given interval, leading to a complex shape (Figure 8.1 shows the shape of the 2-dimensional Z-region [9,17]). This makes it difficult to derive some interesting properties of the spatial object, e.g., minimum and maximum extent in every dimension, etc. As it is common technique in GIS (geographic information systems) and other applications handling extended objects [GG98], we propose to approximate a Z-region by a minimum bounding box (MBB).

An MBB B is a hyper-rectangle, i.e., it is iso-oriented and can be specified by an interval in each dimension. Consequently, an MBB can be specified like a query box (see Definition 2.12) by a multidimensional interval $B = [[a, b]]$.

For computing the MBB B of a Z-region R, we have to determine the minimum and maximum values covered by R in each dimension. Let $min^i(R)$ and $max^i(R)$ denote the minimum resp. maximum value of dimension i covered by R.

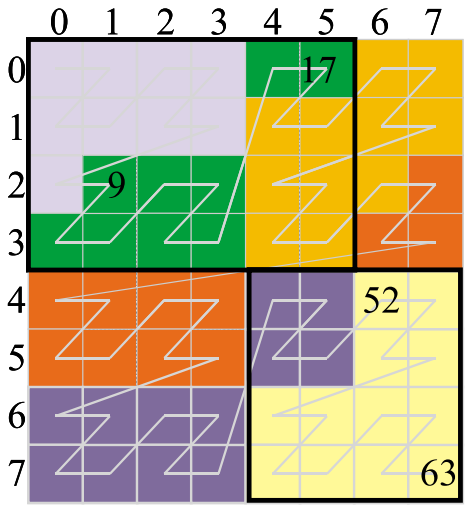


Figure 8.1: Spatial extent of Z-regions [9,17] and [52,63] and the corresponding MBBs

Definition 8.1 (MBB of a Z-region)
A minimum bounding box $B = MBB(R) = [[a, b]]$ *is called the MBB of Z-region* R *iff:*

$$\forall i, 1 \leq i \leq d : a_i = min^i(R) \wedge b_i = max^i(R)$$

Given this definition, the question remains how to compute $min^i(R)$ and $max^i(R)$. To this end, we regard a Z-region R as an interval, i.e., $R = [\alpha, \beta]$. Figure 8.2 shows the algorithm for computing the MBB of a Z-region, requiring only a complete scan through the two Z-addresses α and β. The basic idea behind this algorithm is to check which minimal and maximal bit values may occur for a Z-address cur with $\alpha \leq cur \leq \beta$. For each bit position p in cur the possible value of cur_p depends on the prefixes of α, β, and cur:

$$\forall p \text{ with } l - 1 \geq p \geq 0 : \alpha_{...p} \leq cur_{...p} \leq \beta_{...p}$$

If we now want to construct the minimal (resp. maximal) value for a dimension i, we have to consider the influence of the other dimensions on the prefix. To get the minimal (resp. maximal) value, we assume the optimal case that all other dimensions have the maximal (resp. minimal) possible value in the prefix. If the modified prefix exceeds α (resp. is below β) the value at the current bit position can be set to 0 (resp. 1), otherwise it gets the corresponding value of α (resp. β). Example 8.1 illustrates the algorithm for one Z-region.

Example 8.1: Z-region approximation

Consider the Z-region R=[α, β]=[9,17]=[001001, 010001] of Figure 8.1 (dimension 1 is oriented vertically; dimension 2 is oriented horizontally). The MBB $[[a,b]] = [\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}, \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}] = [\begin{pmatrix} a_{1_2}a_{1_1}a_{1_0} \\ a_{2_2}a_{2_1}a_{2_0} \end{pmatrix}, \begin{pmatrix} b_{1_2}b_{1_1}b_{1_0} \\ b_{2_2}b_{2_1}b_{2_0} \end{pmatrix}]$ of R is computed as follows.

a and b are constructed iteratively by processing α and β starting with the most significant bits. The bits that have not been determined yet are denoted by $'*'$s. As described above, in each step two temporary prefixes *min-tmp* and *max-tmp* are constructed and compared to α and β. Comparing *min-tmp* to α yields the minimum of the MBB, while comparing *max-tmp* to β yields the maximum.

1. Step: handling dimension 1
   The first step is trivial, no comparison is actually needed.
   $min\text{-}tmp = \alpha_5 = 0 \Rightarrow a_1 = 0**$
   $max\text{-}tmp = \beta_5 = 0 \Rightarrow b_1 = 0**$

2. Step: handling dimension 2
   To get *min-tmp* we take the value of the maximum of the MBB for dimension 1, i.e., b_1 and the value for the minimum for dimension 2. Analogously, we create *max-tmp*. The last bit is always taken from α resp. β.
   $min\text{-}tmp = b_{1_2}\alpha_4 = 00;\ min\text{-}tmp = \alpha_{...4} \Rightarrow a_2 = 0**$
   $max\text{-}tmp = a_{1_2}\beta_4 = 01;\ max\text{-}tmp = \beta_{...4} \Rightarrow b_2 = 1**$

3. Step: handling dimension 1
   As soon as the temporary prefix is larger than α (resp. smaller than β) we can stop the computation for the corresponding dimension and fill-up the remaining bits of a and b with 1s resp. 0s.
   $min\text{-}tmp = a_{1_2}b_{2_2}\alpha_3 = 011;\ min\text{-}tmp > \alpha_{...3} \Rightarrow a_1 = 000$
   $max\text{-}tmp = b_{1_2}a_{2_2}\beta_3 = 000;\ max\text{-}tmp < \beta_{...3} \Rightarrow b_1 = 011$

4. Step: handling dimension 2
   $min\text{-}tmp = b_{1_2}a_{2_2}b_{1_1}\alpha_2 = 0010;\ min\text{-}tmp = \alpha_{...2} \Rightarrow a_2 = 00*$
   $max\text{-}tmp = a_{1_2}b_{2_1}a_{1_1}\beta_2 = 0100;\ max\text{-}tmp = \beta_{...2} \Rightarrow b_2 = 10*$

5. Step: handling dimension 2
   $min\text{-}tmp = b_{1_2}a_{2_2}b_{1_1}a_{2_1}b_{1_0}\alpha_0 = 001010;\ min\text{-}tmp > \alpha \Rightarrow a_2 = 000$
   $max\text{-}tmp = a_{1_2}b_{2_2}a_{1_1}b_{2_1}a_{1_0}\beta_0 = 010000;\ max\text{-}tmp < \beta \Rightarrow b_2 = 101$

$\Longrightarrow$MBB(R)=$[\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 3 \\ 5 \end{pmatrix}]$

$\diamond$

For the computation we have to introduce the prefix function $pre(a, s)$ that returns the first s bits of the bit representation of a. If $s = 0$, the result of $pre(a, s)$ is an empty bit string and we extent the comparison function on bit strings to such a case by defining two empty bit strings to be equal. For the algorithm we have to define one more function *dim_replace* that sets the bits of one specific dimension i in a Z-address α according to another Z-address β. More formally,

$$dim\_replace : \mathcal{Z} \times \mathcal{Z} \times \mathbb{N}_0 \rightarrow \mathcal{Z}$$
$$\gamma = dim\_replace(\alpha, \beta, i) \text{ with } \gamma_p = \begin{cases} \beta_p, & \text{if } dim(\gamma, p) = i \\ \alpha_p, & \text{else} \end{cases}$$

```
1:  ZMBB(α,β, zmin, zmax) {
2:     int i;
3:     Z-value min_tmp;
4:     Z-value max_tmp;
5:
6:     for(i=l-1;i=0;i--) {
7:        min_tmp=dim_replace(zmax,zmin,dim(α,i));
8:        max_tmp=dim_replace(zmin,zmax,dim(α,i));
9:
10:       if(pre(min_tmp,l-1-i) > pre(α,l-1-i))
11:          zmin_i = 0;
12:       else
13:          zmin_i = α_i;
14:
15:       if(pre(max_tmp,l-1-i) < pre(β,l-1-i))
16:          zmax_i = 1;
17:       else
18:          zmax_i = β_i;
19:    }
20: }
```

Figure 8.2: Computing the MBB for a Z-region

The result of the ZMBB-algorithm are two Z-values *zmin* and *zmax* containing the minimum resp. maximum values covered by the region in each dimension. In order to get the final MBB we have to extract individual dimension values from those Z-values, that is, the results is the bounding box $B = [[min, max]]$ with $min_i = zmin^i$ and $max_i = zmax^i$.

Complexity of computing the MBB

For computing the MBB of a Z-region we need to compare l times the prefixes of α, β, and *cur*. More precisely, we need $2\sum_{i=0}^{l-1} i = 2\frac{(l-1)*l}{2} = l^2 - l$ bit comparisons and in addition $2 * l$ write operations to create the result.

8.2 Quality of MBB approximation

As with any approximation, the question arises about the quality of the MBB approximation of Z-regions. For the Z-regions in a UB-Tree we expect good estimation by MBBs as the splitting algorithm of the UB-Tree (cf. Section 6.1.3) tries to create as rectangular Z-regions as possible. A frequent method for measuring the quality of an approximation is to compare the volumes of the original and the approximate object. As we are working with discrete domains, the volume of a Z-region is given by the length of the corresponding Z-interval.

Definition 8.2 (Volume of a Z-region)
The volume $vol(R)$ of a Z-region $R = [\alpha, \beta]$ is given by the length of the interval $[\alpha, \beta]$, i.e., $vol(R) = \beta - \alpha + 1$.

Analogously, we define the volume of the MBB of a Z-region as follows.

Definition 8.3 (Volume of an MBB)
The volume $vol(B)$ of a d-dimensional MBB $B = [[a, b]]$ is the product of the lengths of the d intervals $[a_i, b_i]$, i.e., $vol(B) = \prod_{i=1}^{d}(b_i - a_i + 1)$.

Definition 8.4 (Approximation quality)
The quality of the approximation of a Z-region R by the corresponding MBB $B(R)$ is $\kappa(R) = \frac{vol(R)}{vol(B(R))}$.

Table 8.1 provides the quality of the approximation for some of our test data sets.

Table 8.1: Approximation quality for 1 MBB

Data Set	min(κ)	avg(κ)	max(κ)
GEO	67%	82%	100%
CENSUS5D	50%	71%	100%
GfK3D	58%	75%	100%

8.3 Optimization issues

The quality of the approximation can be improved by using more MBBs to approximate one Z-region. This can be achieved by partitioning the Z-regions into a set of disjoint regions. This is especially beneficial for so-called *jump regions*, which decompose in (at most) two disjoint sets of points. Those jump regions are rare, but lead to large, imprecise MBBs (cf. Z-region [9,17] in Figure 8.1). Generally, the best way to split Z-regions is to use the UB-Tree split algorithm which already tries to create rectangular Z-regions. Table 8.2 shows the approximation quality for the approximation with 2 and 4 bounding boxes.

The results show a drastic improvement of the quality compared to just using one MBB for a Z-region. With an average quality $\kappa > 90\%$ for 2 MBBs resp. $\kappa = 98\%$ for 4 MBBs one achieves a reliable approximation with a limited overhead for handling more MBBs.

Table 8.2: Approximation quality for multiple MBBs

Data Set	**# MBBs**	**min(κ)**	**avg(κ)**	**max(κ)**
GEO	2	83%	93%	100%
GEO	4	92%	98%	100%
CENSUS5D	2	77%	91%	100%
CENSUS5D	4	89%	98%	100%
GfK3D	2	78%	90%	100%
GfK3D	4	90%	98%	100%

8.4 Chapter notes and related work

In this chapter we have introduced an algorithm for computing the minimum bounding box for a Z-region. This gives the possibility to transfer extended objects in the linear space, i.e., intervals on the Z-curve, back to objects in the multidimensional space. The MBB is not the exact description of the object but an approximation that is much easier to describe. Our experimental results show that a small number of MBBS leads already to a very good quality of the approximation. Thus, this approach is very useful for problems where the exact Z-region is not required, for example, for visualization, approximate query processing, analysis of data distribution and much more.

Advanced Nearest Neighbor Search on UB-Trees Nearest Neighbor (NN) search and its derivatives (k-NN, approximate NN, etc.) have become very important in various application areas in recent years [RKV95]. In [Mar99, Str00] two NN algorithms for UB-Trees are proposed and analyzed. One of the key issues in these algorithms is the computation of the distance from a point to the Z-region boundaries in order to detect the next Z-region to be processed. The MBB-approximation will drastically simplify the distance computation but with the disadvantage that it is only an estimation. Still, this approach could be viable for approximate NN-Search on UB-Trees.

Tetris-Algorithm In the Tetris algorithm there are two places where the MBB approximation may help. First, in the computation of the next event point and secondly to determine all Z-regions behind the sweep line. These tests require the maximal extension of a region for a given dimension, which is naturally provided by the MBB of the Z-region.

Related work

The problem of approximating a Z-region with a (set of) MBBs can be regarded to be the inverse problem of approximating extended objects by a set of intervals on space-filling curves [Gae95, GG98].

Chapter 9

Query optimization in the presence of UB-Trees

As already pointed out in Chapter 6, when we are integrating a new data structure into a DBMS kernel, we also have to guarantee that it can effectively be used by the query optimizer.

In this chapter, we discuss how the availability of UB-Trees affects modern optimizers. We address the issue of new execution possibilities given by the new access method and dilate on the important topic of cardinality estimation, which is essential for cost-based optimization.

9.1 Introduction to query optimization

In modern DBMSs, query processing is generally divided into six steps [Gra93]:

1. Parsing
2. Query Validation
3. View Resolution
4. Optimization
5. Plan Compilation
6. Query Execution

After a query is issued to the DBMS, it is syntactically checked and parsed into an internal form (Step1). In Steps 2 and 3, the query is semantically checked (validated) against the meta data (catalog) and, if necessary, view definitions and subqueries are resolved. Often some rule-based transformation (sometimes called *query rewrite*) is already applied in Step 3. This only includes optimizations that are independent of later cost-based decisions, i.e., optimizations that guarantee to always produce better query plans. Optimizations, like decisions on access paths or join-orders, that depend heavily on the data distribution of the base tables are

usually left to the cost-based optimizer. Query rewrites that are applied in this phase usually consist of safe relational algebra transformations such as pushing restrictions and projections down as far as possible.

In the subsequent optimization step, an optimized query evaluation plan (QEP) is generated. This requires the mapping of *logical* operators to *physical* operators, i.e., concrete implementations of logical operators (e.g., hash-join implementation of an equi-join). Using cost estimations the optimizer chooses the best QEP from all possible ones. This QEP is then translated/compiled into an execution ready form and finally executed in the query engine.

The optimization phase itself can be further divided into *plan generation* and *cost estimation* [Cha98].

Plan generation

The plan generator enumerates potential QEPs for a given logical plan. For each generated plan the optimizer then estimates the cost and chooses the best one for execution. Typically, the two steps are interleaved as the results of the cost estimation help to reduce the *search space*, i.e., the number of plans which have to be considered by the plan generator. Integrating a new index structure may also introduce new operators that have to be taken into account.

Cost estimation

In order to choose the best (or, to be more precise, a reasonably efficient) plan the optimizer needs to estimate the cost of each plan produced by the plan generator. To this end, the cost of each operator in the plan has to be guessed. In modern optimizer models, the cost of an operator is usually determined by the size of its input, i.e., by the number of tuples processed by the operator. Consequently, the optimizer requires reliable cardinality estimates for all outputs of operators at each level of the plan. The same holds for the access operators of base tables: the I/O cost is estimated by a cardinality estimation for the predicate on the base table multiplied by an individual factor, reflecting the I/O overhead, for each access method. The cardinality estimation on the base table is usually done independently of the available index structures to prevent biased cost estimates.

In the next section, we discuss the changes to the optimizer triggered by the UB-Tree integration.

9.2 Rewrite rules for UB-Trees

Considering the integration of UB-Trees, the question arises, if there are scenarios in which a UB-Tree access is always favorable compared to all other available methods? Envisioning the vast number of possible execution plans for one query, it is hard to identify situations where one specific plan/operator always outperforms all alternatives. To answer the question if such a situation exists for the UB-Tree, we have to look at the cost of the various index operations again.

Point queries

For point queries we can guarantee constant cost (one data page access) independent of the data distribution under the assumption that the index key is unique. If there are duplicates, even point queries may lead to multiple data page accesses, depending on the number of duplicates and the data distribution. Still, it is relatively safe to assume that duplicates do not cause a major problem. So, given a point query, is the UB-Tree always the best choice? No, as it depends on the query and on the alternatives. For example, assume a second, non-clustering index, with the same key that also *covers* all projection attributes required by the query. It may be cheaper to use that one, because if this non-clustering index does not include all attributes of the base table it is usually smaller and thus cheaper to access.

There is one safe rewrite rule for access methods in general that is already used in systems. Informally, applied to the UB-Tree is states: if the query restricts all index keys to a point and if the UB-Tree is the only index that covers all required attributes then use the UB-Tree.

Range queries

For range queries no prediction on the access cost can be made without looking at the data or reliable statistics. Without a cost estimation it is therefore not possible to judge which access method yields the lowest cost. Even if there are no other indexes, it is not safe to use the UB-Tree. Recalling the rule of thumb stating that a full table scan outperforms any index if more than 10% of the data has to be fetched, even a full table scan may be the better choice for range queries. Thus, the decision on the access method should be left to the cost-based optimizer.

We conclude that the general rules for access methods suffice, and that we have not identified special transformations for the UB-Tree that can be safely applied in the rewrite phase.

9.3 Plan generation for UB-Trees

Supporting the UB-Tree as multidimensional access method not only provides a new access path but also gives room for new execution plans. In the following we briefly describe new execution plan variants that should be considered by the optimizer:

- Star join optimization
- Z-order join for Equi-Joins
- Combination of range query and sorting
- Optimization for multiple query boxes
- Reduction of post-filtering

It is important to note, that we discuss new *potential* execution plans here. The decision if a new execution variant is actually better than others should still be done by the cost estimation.

Star Joins with UB-Trees

Star join optimization plays an important role for data warehousing applications. Much work has been done in the past to optimize star-join queries leading to various hash-based [Sun96] or index-based [OQ97] approaches and combinations of them [ZDNS98].

As for bitmap indexes, the idea of index-based star joins can also be used for UB-Trees or any multidimensional access method in general: instead of joining the complete fact table with the dimension tables, a multidimensional range query on the fact table is used to reduce the size of the largest join partner before applying any other star join technique on the result. Star join queries usually include local restrictions on the participating dimension tables. The UB-Tree allows for utilizing these local restrictions to reduce the size of the fact table before joining it with the dimension tables. To this end, a plan has to be generated that maps the local restrictions to restrictions on the dimension keys, then combines these restrictions to one or more multidimensional queries on the fact table. Such a plan is especially advantageous if the resulting set of multidimensional queries on the fact table is small. The goal of this step is to reduce the fact table before the costly joins with the dimension tables for which the optimizer may choose any available method.

Z-order join

The Z-order of UB-Trees can be utilized in processing of equi-joins. If all participating tables have a UB-Tree on the join attributes and the bit-interleaving is in the same order, then the join can be computed by just merging the corresponding Z-values, utilizing the underlying order of UB-Trees.

Multiple query boxes

As we have mentioned in Section 6.3, the UB-Tree also handles a set of multidimensional query boxes very well. Yet, if the number of query boxes gets very large, the overhead of handling all these boxes may become intolerably high. For such cases, it may be more beneficial to use one or more larger query boxes that cover the set of query boxes in combination with appropriate post filtering. There are two ways to generate the covering query boxes: first, one can use an algorithm to find a good cover given the complete set of original query boxes. This approach, however, requires dynamic optimization, i.e., optimization at query execution, as it needs the complete set of query boxes, which are unlikely to be specified in the original query itself but are generated by the evaluation of some other predicates. Secondly, one has to change the generation of the original query box set. Like in the star-join case mentioned above, the query box set is usually the result of the evaluation of local predicates on the dimensions. Such predicates lead in general to a set of ranges on the dimension keys. The multidimensional query boxes are the

result of the cross-product of all these key ranges. Consequently, a large set of query boxes is created if the local predicates generate many key ranges/intervals on the dimensions. If the optimizer can detect statically that a given dimension predicate may lead to many intervals it may reduce it for the query box generation in combination with a subsequent post-filtering. For example, reducing the predicate "every second day in 2002" on the time dimension to "all days in 2002" (resulting in just one range) reduces the number of query boxes by the factor 182.

Combination of sorting and range queries

Sorting plays an important role in query processing. Either because it is requested by the user or because it is required or useful for subsequent internal operators (e.g., sort-merge join, grouping, duplicate elimination). The Tetris-Algorithm for the UB-Tree [MZB99] enables the combination of sorting and range query processing leading to new alternatives to be considered by the plan generator. In combination with the TempTris-Algorithm [ZMB01], the Tetris-Algorithm can be used to organize large intermediate results as UB-Trees. This allows for efficient multidimensional access to intermediate results if needed.

Reduction of post-filtering

If a query restricts both, key and non-key attributes, usually a postfiltering step for the non-key predicates is required. The range query algorithm of the UB-Tree can easily be extended to combine these two steps: instead of just using the query box specification in the `postFilterPage` function (cf. Figure 7.12), we can additionally check the non-key predicates. This procedure corresponds to the two-phase processing in many spatial indexes, where we distinguish between a *filter* step and a *refinement* step [RSV01]. This optimization is especially beneficial for clustering UB-Trees because in this case the non-key attributes are basically at hand.

9.4 Cardinality estimation

As we have stated before, the crucial issue of cost-based optimization is to get good estimates for the cardinalities of operator outputs. The query optimizer needs this information for two reasons:

1. Estimation of costs: the result set size can be seen as an estimate for the costs of producing the result

2. Decision on the order of operators: decision of join orders and others

For the further discussion it is necessary to clarify the two terms cardinality and selectivity. Cardinality refers to the size of a set of tuples, either the input or output of an operator. As defined in Section 2.3, selectivity refers to the restrictiveness of a query/perdicate, i.e., the ratio of the output over the input of a restriction operator. The techniques described below all estimate the cardinality of the result of a query

and not directly the selectivity. Still, the selectivity can be easily derived from the cardinality estimate and the cardinality of the complete table.

In this section, we present a new histogram-based estimation technique for multidimensional data relying on UB-Tree techniques. We also point out the limits of histogram-based estimation and suggest a new quality metric for multidimensional histograms.

9.4.1 State of the art estimation techniques

Before directly going to the description of the new estimation techniques, we give a brief overview on the state of the art of cardinality estimation. We are concentrating on histogram-based techniques here and discuss other methods in Section 9.5.

Histograms are a synopsis of the data distribution that partition the attribute domain into a set of buckets. In general, data is assumed to be uniformly distributed within a bucket and the distribution of the buckets approximates the real data distribution. Each bucket b stores the description which subset of the domain is covered and the frequency $freq^b$ specifying the number of tuples within this subspace. The so-called *uniformity assumption* provides the foundation for histogram-based estimation: it states that the data is uniformly distributed within a bucket. Thus, given a query Q and a histogram with n buckets $b_1, \ldots, b_n$, the cardinality of Q is estimated as:

$$|Q|^{est} = \sum_{i=1}^{n} \left(freq^{b_i} * \frac{vol(Q \cap b_i)}{vol(b_i)} \right)$$

with $Q \cap b_i$ specifying the intersection of Q with bucket b_i.

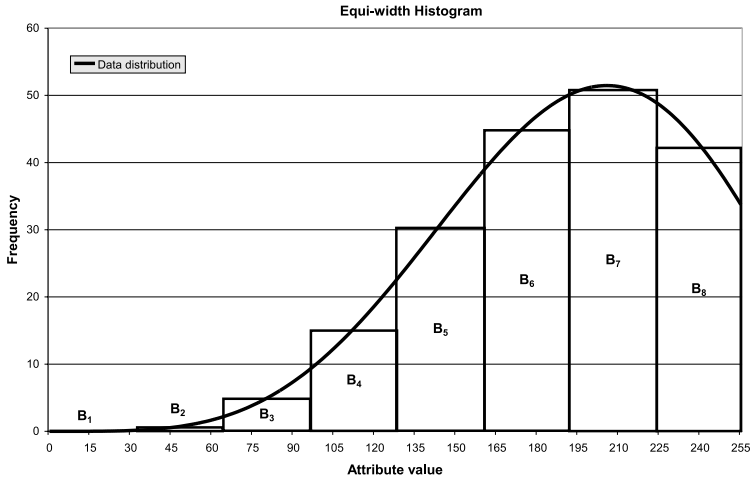
The precision of a histogram depends largely on the number of buckets in the histogram: the more buckets the better the approximation of the real data distribution. On the other hand, more buckets lead to higher space consumption and estimation cost as more buckets need to be stored and processed. In modern DBMSs the number of buckets is usually configurable. Commonly, 2KB ($\approx$ 512 buckets) are used per attribute.

The uniformity assumption for buckets is the limiting factor for the usage of histograms as it may lead to useless estimations in very sparse universes as we discuss in Section 9.4.3.

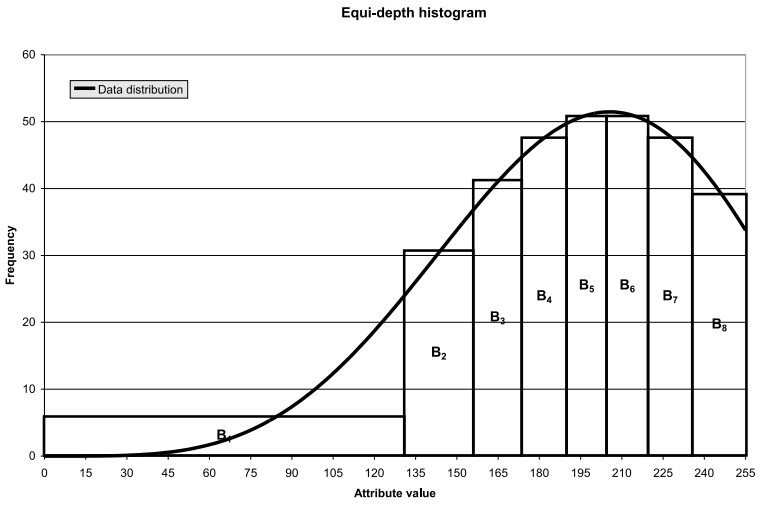
9.4.1.1 One dimensional histograms

One dimensional histograms are used to approximate the data distribution in one attribute/dimension. Two simple histogram types are depicted in Figure 9.1.

Figure 9.1(a) shows a *equi-width* histogram where the complete data space is divided into n buckets of the same length. The *equi-depth* histogram (Figure 9.1(b)), in contrast, divides the data space into buckets that have the same frequency. Consequently, dense data areas are covered by many small buckets whereas sparse areas are covered by few large buckets. In general, equi-depth histograms are preferred over equi-width histograms as they adapt better to ragged data distributions.



(a) Equi-width histogram



(b) Equi-depth histogram

Figure 9.1: Equi-width and Equi-depth histograms

Over the time, these simple histograms have been tuned and enhanced, e.g., V-Optimal histograms [IP95], to provide faster and better estimations.

9.4.1.2 Multidimensional histograms

One dimensional synopses cover one attribute at a time. If one wants to approximate the joint data distribution of multiple attributes, one-dimensional histograms do not suffice, unless the attributes are completely independent from each other. If this *independence assumption* holds, the overall selectivity of query of a conjunction of local predicates σ_{A_i} on attribute A_i is the product of the individual selectivities, i.e.,

$$sel(\sigma_{A_1} \wedge \ldots \wedge \sigma_{A_d}) = sel(\sigma_{A_1}) * \ldots * sel(\sigma_{A_d}).$$

The problem of this approach lies in the fact that the independence assumption usually does not hold in practice. Consequently, the estimation based on the above formula may lead to high error rates for data distributions with correlations.

Recognizing this issue, researches recently have come up with various multidimensional histograms explicitly capturing the joint data distribution of attributes. [GG01] provides an excellent overview on the latest developments. The problem of multidimensional histograms is comparable to the one of multidimensional indexes: finding a good bucketization, i.e., partitioning of the multidimensional domain into buckets, to approximate the real data distribution as exactly as possible. The approaches range from simple multidimensional equi-depth histograms [MD88] to histograms allowing for overlapping buckets [GKTD00]. The generation and maintenance of multidimensional histograms is quite expensive and their prediction precision deteriorates with growing universe size (see Section 9.4.3 for details). Recent work tries to optimize the histograms by exploiting knowledge about the independence and correlation of attributes to reduce the dimensionality [DGR01].

9.4.1.3 Index-based cardinality estimation

Using histograms for cardinality estimation has a significant drawback: histograms are static. A histogram always reflects the data distribution at the time of histogram creation. Usually, it is too expensive to update histograms dynamically. As consequence, the precision of histograms deteriorates with the number of insert, delete, and update operations executed on the underlying table. Thus, it is necessary to periodically gather the statistics from scratch to achieve tolerable error rates. Such a refresh of the histogram is usually done in the maintenance period of the database but still requires a significant amount of time.

To get more accurate and actual statistics, various approaches exist to use indexes as a replacement/add-on to histograms. Indexes are often used in combination with sampling and probing [Ant92], but the index can also be regarded as an equi-depth histogram itself [Aok98a]. The technique of using indexes for estimation is already available in commercial DBMSs, e.g., Oracle [AZ96]. An additional advantage of indexes in comparison to histograms is the non-fixed resolution. If necessary, the resolution of the estimation can be improved by traversing deeper levels of the tree.

A problem of index-based cardinality estimation is the danger of biased decisions, if the estimates stemming from indexes are compared to the ones from standard histograms.

9.4.1.4 Estimation error metrics

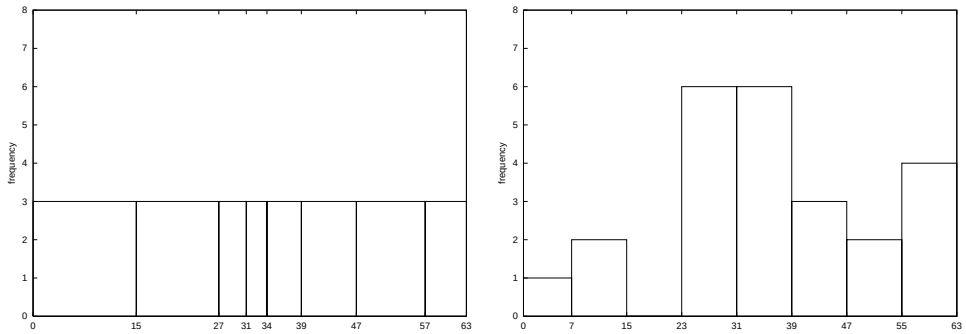
For the experimental evaluation of histograms we will use standard metrics as used in previous work. The goal is to measure the estimation error of the correct result set size $s = |Q|$ and the estimated size $s^{est} = |Q|^{est}$ of a query Q. The most common used metric is the *(absolute) relative error* defined as follows[1]:

$$E^{rel} = \frac{|s^{est} - s|}{max(1, s)}$$

For a set of queries we use the median of E^{rel} of the individual queries.

9.4.2 Cardinality estimation with UB-Tree techniques: $\mathcal{Z}$-Histograms

We have discussed above that the problems of multidimensional histograms are closely related to the issues in multidimensional indexes. With the UB-Tree we have an efficient method for handling multidimensional data at hand. So why not use this technology also for cardinality estimation? Instead of working with multidimensional histograms, we use the combination of dimension-reduction by the Z-curve and one-dimensional histograms. Or to be more precise, we use $\mathcal{Z}$ as the base space of the histograms. Each bucket b is then defined by a Z-interval and consequently represents a Z-region. Figure 9.2 shows an equi-depth and an equi-width histogram for a non-uniform data distribution.



(a) Equi-depth histogram (b) Equi-width histogram

Figure 9.2: Equi-depth and equi-width histogram on Z-values

[1]It is necessary to use the term $max(1, s)$ in the denominator in order to allow for empty queries, i.e., queries that have no result.

In our experiments, we have observed that equi-depth histograms usually perform better than equi-width histograms. Thus, we will only report the results for our measurements with equi-depth histograms in the subsequent sections.

Optimizing $\mathcal{Z}$-Histograms

As the buckets of $\mathcal{Z}$-histograms cover the complete space the problem of covering 'dead space' arises. For equi-depth histograms, however, the dead space covered by a bucket can easily be reduced by using the Z-values of the first and the last tuple falling into the bucket as bucket boundaries. This idea of reducing dead space is currently also applied to reduce the dead space indexed by the UB-Tree (see [Fen02] for details).

9.4.2.1 Estimation based on linearization of query boxes

We discuss the estimation using the linearization of query boxes only briefly as this approach has severe practical limitations with respect to the runtime of the estimation (details can be found in [Tic99]).

The idea is to linearize the query Q into a set of m Z-intervals $Z = \{z_1, \ldots, z_m\}$. For each interval we compute the overlap with the buckets of the $\mathcal{Z}$-histogram and estimate the contribution per bucket with the product of the frequency and the percentage of the overlap.

$$|Q|^{sel} = \sum_{j=0}^{m} \sum_{i=0}^{n} \left(freq^{b_i} * \frac{vol(z_j \cap b_i)}{vol(b_i)} \right)$$

The quality of this method is very good, generating error rates below 1% on various artificial data distributions. However, the price for this accuracy is intolerable high: the estimation took between 6 and 100 times longer than the execution of the query itself. The reason for this drastic difference is the large number of intervals a query box may correspond to (cf. Section 7.2). We tried various ways of reducing the number of intervals by approximating the query box, i.e., trading accuracy for estimation cost. While we were able to reduce the execution time to a tolerable level, we could achieve reasonable estimation results only for up-to three dimensions and mid-size queries ($>$ 0,25 % selectivity).

9.4.2.2 Estimation based on MBB-approximation of Z-regions

Instead of linearizing the query box and therefore working in one-dimensional space, we now approximate each bucket by the corresponding MBB (see Chapter 8 for details) and work in the multidimensional universe to estimate the cardinality. To this end, we compute for each bucket b of our $\mathcal{Z}$-histogram the MBB $B(b)$ and check whether it intersects the query Q. We can use the standard estimation formula, i.e.,

$$|Q|^{sel} = \sum_{i}^{n} \left(freq^{b_i} * \frac{vol(Q \cap B(b_i))}{vol(B(b_i))} \right)$$

This estimation is by far faster than the estimation based on the linearization of the query box, as at most n buckets have to be tested against Q. The accuracy of this technique is good, matching the ones of other multidimensional histograms. Table 9.1 shows the estimation error for a set of queries on the example data sets. We classify the query sets into four groups according to the selectivity of the queries. The histogram size, i.e., the number of buckets, was chosen to be approximately 2KB. We show the median relative estimation error E^{rel} for each group and the standard deviation.

Table 9.1: Estimation results

Data Set	**# buckets**	**Query selectivity**	**median** E^{rel}	**stdv**(E^{rel})
GEO	300	<0,1%	200%	5167%
GEO	300	0,1% - 1%	21%	25%
GEO	300	1% - 10%	6%	7%
GEO	300	> 10%	2%	2%
CENSUS5D	600	<0,1%	678%	6599%
CENSUS5D	600	0,1% - 1%	78%	91%
CENSUS5D	600	1% - 10%	11%	22%
CENSUS5D	600	> 10%	7%	4%

The results of Table 9.1 show the $\mathcal{Z}$-Histograms work very well for queries with a selectivity of more than 1%. For smaller queries the estimation error becomes larger, getting especially high for queries with a selectivity < 0,1%. But even this high errors are tolerable in practice, as the overall estimated cardinality still represents a small result, providing a reliable hint for the optimizer (e.g., it does not matter if the optimizer calculates with 500 tuples instead of 1 tuple, i.e., $E^{rel} \approx 500\%$).

For the GfK data sets, however, the estimations become completely unreliable, returning a cardinality of 0 for all queries. In the next section we discuss this, in our opinion, general problem of histograms with sparse universes in more detail. In the meantime we conclude that the estimation based on MBB-approximation of Z-regions works well if the underlying histogram is reliable.

9.4.3 When histogram based cardinality estimation fails

Now, we want to discuss the limits of histogram based cardinality estimation. As we have pointed out in the discussion above, the uniformity assumption for buckets is the key concept of histogram based estimation. We investigate under which circumstances this assumption does not hold and hence the histogram estimation will lead to intolerable errors. In the literature, the infamous "curse of dimensionality" is often stated as key reason for this problem. We evaluate this more precisely and come to the conclusion that it should be better called *"curse of sparse universes"*.

Curse of dimensionality

The term "curse of dimensionality" usually refers to the problem of distance metrics in high-dimensional data spaces [WSB98]. With increasing dimensionality the number of direct neighbors for a given point increases linearly and in general the number of points within a given distance around the search point increases exponentially. As consequence, standard distance metrics become almost meaningless in those spaces. The consequences for multidimensional access methods are quite obvious: the clustering of those index structures deteriorates with increasing dimensionality.

The question that arises is, how does the dimensionality affect the quality of histograms? As introduced earlier, histograms are a synopsis or summary of the (populated) universe Ω. Each bucket covers a certain part of Ω and within the bucket the data is assumed to be uniformly distributed. Again, lets have a look at the estimation formula for a query Q:

$$|Q|^{est} = \sum_{i}^{n} \left(freq^{b_i} * \frac{vol(Q \cap b_i)}{vol(b_i)} \right)$$

Analyzing when this formula leads to $|Q|^{est} \approx 0$ results in:

$$|Q|^{est} \longrightarrow 0 \quad \text{if} \quad \forall b_i : freq^{b_i} * \frac{vol(Q \cap b_i)}{vol(b_i)} \longrightarrow 0.$$

This can only be the case if the intersection of the query with the bucket is much much smaller than the bucket itself, i.e., if $\frac{vol(Q \cap b_i)}{vol(b_i)} \longrightarrow 0$, as in general $freq^{b_i} > 0$ holds for all b_i.

We will discuss the influence factors that may lead to this effect. As we will see, the problem exists for large and very sparse universes as they are typical for real-world applications.

1. Observation: Size of universe has more influence than dimensionality

The size of the d-dimensional universe $\Omega = \mathbb{D}_1 \times \ldots \times \mathbb{D}_d$ is $|\Omega| = \prod_{i=1}^{d} |\mathbb{D}_i|$. Assuming the same domain $\mathbb{D}$ with size $s = |\mathbb{D}|$ for all dimensions then $|\Omega| = s^d$. Table 9.2 illustrates the different influence of the domain size and the dimensionality on the universe size. For example, the two-dimensional universe based on the integer domain (2^{31}) is larger than the 6-dimensional universe based on a domain of size 1024. Hence, there is definitely a "curse of dimensionality" as each added dimension also expands the universe, but a more important influence factor on the performance of histograms is the resulting *universe size*.

2. Observation: Universes are very sparse

To our surprise, the sparsity of real-world data universes is often underestimated in the literature. For example, [Col96] calls 80% sparsity typical for OLAP applications, a value also mentioned by other publications [NNT00]. In our own experiments

Table 9.2: Universe size: dimensionality vs. domain size

s	d	$\|\Omega\|$
1024	2	1048576
1024	4	1,09951E+12
1024	6	1,15292E+18
1024	8	1,20893E+24
2^{31}	2	4,61169E+18
2^{31}	4	2,12676E+37
2^{31}	6	9,80797E+55
2^{31}	8	4,52313E+74

with various real-world data sets (cf. Chapter 4), however, we have observed an occupation less than 1%, i.e., a sparsity significantly higher than 99%, to be typical. Example 9.1 presents the sparsity of the 3-dimensional GfK data set.

Example 9.1: Sparsity

We will use our 3-dimensional GfK data warehouse to illustrate the problem of large universes. As we have described in Chapter 4, the universe of GfK is very sparse. With a product domain of size 2^{29}, a segment domain of 2^{24}, and a time domain of 2^5, the complete universe covers 2^{58} points. With a data volume of approx. 43 ($\approx 2,5 * 2^{24}$) million tuples the sparsity is larger than 99,99999998%, i.e., less than 0,00000002% of the universe is populated.

◇

In our opinion, the very sparse universes in real applications stem from the huge difference between the sizes of the actual domain and the nominal domain of dimensions. Even though the actual domain, i.e., the existing values of the attribute, is usually small (e.g., 500.000 products for GfK) the nominal domain is usually much larger (e.g., integer domain without constraints in GfK for the Product dimension; the same holds for the TPC-H [TPC99] benchmark: for the ORDER table a sparsity of 25% is specified leading to a similar sparsity for the LINEITEM table as for GfK). This is due to the fact that mostly artificial keys, which often also carry some semantics, are used causing a wide spread of the keys leading to a large nominal domain.

It is important to note in this context, that the sparsity heavily depends on the type of data. Raster data, like X-Rays, CT, or MRI data, is a typical example of dense multidimensional data with a sparsity close or equal to 0.

3. Observation: Data is widely spread resulting in sparse buckets

The difference between nominal and actual domain alone is not the key problem of histograms. If the data would reside in few data clusters, which have relatively

small distance from each other, histograms would be able to approximate this data distribution very accurately. However, many real-world examples show that a huge difference between actual and nominal domain is often the result of widely spread data in the attribute. In combination with other dimensions this leads to a large number of clusters in the multidimensional universe. If the number of clusters exceeds the number of buckets available for approximating the data distribution, multiple data clusters have to be covered by one bucket. Depending on the number of clusters in one bucket and the distance between these clusters the area covered by a bucket may get very large. Consequently, the sparsity of buckets (see Definition 9.1) may get very high.

4. Observation: Queries are tiny

User queries usually focus on some data cluster, e.g., an analyst will not ask for the sales from 1970 to 2000 if she knows that the data starts only in 1990. Consequently, queries commonly have at most the size of a data cluster.

Putting all observations together, we can state that histograms have problems if

- the universe is sparse and
- there are more data clusters than buckets and
- queries are much smaller than buckets, i.e., $\frac{vol(Q \cap b_i)}{vol(b_i)} \longrightarrow 0$.

This is for example the case for the GfK application and other real-world applications we have examined. Consequently, the question arises how to determine if a histogram will provide reliable estimates or not. As this depends largely on the data distribution to be approximated, one either needs a thorough data analysis before histogram creation or a way to measure the quality of a histogram after creation.

A quality measure for histograms

We suggest to use the *sparsity of buckets* to measure the quality of a histogram.

Definition 9.1 (Sparsity of buckets)
The sparsity ξ of a bucket b is $\xi(b) = 1 - \frac{freq^b}{vol(b)}$.

During or after histogram creation, the sparsity criteria can be either used to judge the quality of the histogram or for further optimization of the histogram. In the first case, a histogram is less reliable the larger the average sparsity of all buckets. To be more precise, only estimates for queries using solely buckets with acceptable sparsity will provide reliable results[2]. Including buckets with high sparsity into the estimation will significantly increase the error rate. On the other side, the sparsity of the buckets can be used for histogram optimization as they indicate where the

[2]It is not scope of this thesis to conduct a quantitative analysis in order to determine a boundary for acceptable sparsity.

approximation has to be improved. Improving the synopsis usually only works by introducing new buckets to get a finer approximation of the data set. The sparsity of the buckets helps to identify areas of the histogram where extra buckets should be spent.

9.5 Chapter notes and related work

In this chapter we have discussed changes of the query optimizer which are required to efficiently support UB-Trees. We have identified two issues: first, the generation of execution plans that take UB-Trees into account. Secondly, an exact and reliable cardinality estimation for predicates on multidimensional data is essential for cost-based optimization.

The modular design of modern optimizers allows for easy integration of new access methods: the plan generator includes new operators and access methods in the creation of potential plans, which are evaluated by the cost estimator. Getting exact and reliable cost estimation for the new index structure is the key issue of the integration. It is important to note that this is not a special issue of the UB-Tree integration, but holds for supporting multidimensional access methods in general.

Modern DBMSs use histogram-based or sampling techniques for cardinality estimation, but the estimation of joint data distribution over multiple attributes is still an open problem. We have proposed $\mathcal{Z}$-Histograms, a new multidimensional histogram technique, combining the Z-curve with one-dimensional histograms. We have examined two ways of approximating the cardinality of a query: first, linearizing the query into a set of intervals. Secondly, transforming the buckets of the histogram, i.e., the Z-regions defined by the bucket, into MBBs using the algorithm of Chapter 8. While the first approach provides excellent precision it leads to intolerable estimation time. The latter approach combines tolerable overhead with usable error rates, comparable to other multidimensional histograms.

Finally, we have pointed out the limitations of histograms in general: the uniformity assumption for buckets. We have shown that this assumption does not hold for sparse universes with scattered data leading to sparse buckets. In such cases, histograms will significantly underestimate the real cardinality, resulting in unreliable cardinality estimations.

This deficit of histograms, which also holds for other estimation techniques like sampling, has lead to further developments. For example, STHoles [BCG01] is a work-load aware histogram, trying to improve the precision of histograms in the *area of interest* of the user, i.e., in the subspace defined by the queries. Such approaches have problems with varying interest, i.e., wide spread queries, and with the estimation of queries falling outside the current focus of the histogram.

A similar approach is LEO, the learning optimizer of IBM DB2 [SLMK01]. The DBMS detects errors in the estimation and stores adjustments resulting from the actual query execution to be used in future optimizations. This corresponds to building-up a histogram for the executed queries and has the same benefits and draw-backs of the previous approach: exact estimates for similar queries but high

error rates for 'outlier' queries. In addition, the precision of the approach also depends on the overhead allowed for storing the adjustments.

Related work

The field of query optimization covers a large body of work such we only highlight contributions important for our work.

Cardinality Estimation Cardinality estimation has a long tradition in query optimization [SAC+79]. Lately, it has received new attention in the context of approximate query processing [GG01], which is considered to be the promising approach to handle the increasing data volumes of today's applications.

[PIHS96] and [Poo97] provide a detailed overview of various histograms and estimation algorithms. [Rit99] evaluates histograms for the usage with raster data. [DGR01] proposes the usage of dependency models to improve the precision of histograms in high-dimensional spaces. [Aok98a, Aok99] discuss the usage of indexes as histograms. Besides histograms, wavelets [CGRS00] and sampling are commonly used for cardinality estimation.

Query processing with HC in EDITH Query processing in the presence of UB-Trees is currently investigated in the EDITH project [EDI]. The focus in this project is not set on the UB-Tree itself, but on hierarchical clustered data. The hierarchy semantics encoded in the artificial keys (surrogates; see Section 2.5) allows for new processing techniques of typical OLAP queries. For example, pregrouping on surrogates significantly improves the overall response time. The work in EDITH relies on a multidimensional organized, more specifically a UB-Tree indexed, fact table to provide efficient access to the base data. The goal of HC is to map restrictions on dimension hierarchies to range restrictions on the fact table. Still, depending on the predicates, a query may map to a large set of intervals in one dimension, leading to a large set of query boxes on the fact table. At that point, the optimizations introduced in this chapter, i.e., the star-join processing and the handling of multiple query boxes, play an important role.

Part III

Advanced concepts in multidimensional indexing

In the last part of this thesis we will enhance the basic concepts of the UB-Tree, going one step further towards a universal index. We discuss a modified address calculation scheme in order to allow for arbitrary weighting of dimensions. A side-effect of the results is an improved range query processing for standard B-Trees, making it also more suitable for multidimensional indexing. We briefly point out open issues and interesting future work before we conclude our work.

Chapter 10

Weighted dimensions

As we have already discussed in the first part, clustering is a key concept for providing efficient access to data. Access methods provide a wide range of different clustering schemes. A large set of index structures cluster the data according to some sort order of the tuples, e.g., the Z-order for UB-Trees. Other access methods use a combination of various heuristics, making it difficult to express the clustering criteria formally, e.g., the clustering of R*-Trees. The quality of a clustering scheme can only be measured w.r. to a given query set. An important property of a multidimensional clustering scheme is the *symmetry* of the clustering. Informally, a clustering scheme is symmetric if no index attribute is preferred w.r. to query performance. For example, UB-Trees and R*-Trees are considered to be symmetric index structures whereas a composite B-Tree is asymmetric, favoring the first attribute of the key.

Symmetric treatment of all index attributes is desired if no knowledge about preferences in the restrictions to the index attributes is available. However, if preferences on the attributes exist then one wants to exploit them for improving the clustering. In this chapter we introduce the new concept of *weighted dimensions*, enhancing the clustering scheme of UB-Trees to allow for asymmetric handling of attributes. Our evaluation shows that the *weighted* UB-Tree provides better performance for graded/weighted multidimensional range restrictions than the standard UB-Tree.

10.1 The concept of weighted dimensions

Before we talk about the concept of weighted dimensions, we have to specify the goal more precisely. The goal is to adapt the clustering of the UB-Tree to the preferences specified in a set of queries. Typically, not all dimensions have the same importance, i.e., there are attributes that are either more frequently restricted than others (e.g., the Time dimension is almost always restricted in data warehouse queries) or that impose stronger restrictions on the table. A symmetrical index, like the UB-Tree, cannot provide the best clustering for such graded queries. For example, a composite B-Tree will always perform better than a UB-Tree if only the first attribute is restricted. At the same time, unsymmetrical indexes lack the

flexibility of supporting changing access patterns. We will discuss these issues later and will first show how changing the address calculation of UB-Trees from bit-interleaving to bit-permutations leads to weighted dimensions, allowing for better performance for graded queries.

10.1.1 An enhanced address calculation concept for UB-Trees

Let Ω be the multidimensional domain of d dimensions $\mathbb{D}_1, \ldots, \mathbb{D}_d$. In the following, we are only interested in the binary representations of the domain values. For ease of illustration, we assume the same length l for the binary representations for all dimensions. Consequently, we can express a point $x = (x_1, \ldots, x_d) \in \Omega$ by a (d,l)-(binary) matrix

$$X = \begin{pmatrix} x_{1_0} & \ldots & x_{1_{l-1}} \\ \vdots & \ddots & \vdots \\ x_{d_0} & \ldots & x_{d_{l-1}} \end{pmatrix}$$

specifying the binary representation of the point. This presentation is chosen, as x_{ij} denotes the $(j-1)^{th}$ bit[1] of dimension i.

For describing arbitrary bit permutations of a tuple x to a bit string b with $|b| = d * l$ we introduce a (l,d)-*permutation matrix* $\mathcal{P}$. The element $\mathcal{P}_{ij}$ specifies to which bit b_k the $(i-1)^{th}$ bit of dimension j maps to (denoted by $x_{j_{i-1}} \mapsto b_k$). Instead of storing the position k in the element $\mathcal{P}_{ij}$, we store the value of b_k, i.e., $\mathcal{P}_{ij} = 2^k \Leftrightarrow x_{j_{i-1}} \mapsto b_k$. With this introduction, we can formally define the permutation matrix as follows.

Definition 10.1 (permutation matrix)
*$\mathfrak{P}$ denotes the set of all permutation matrices. A **permutation** matrix $\mathcal{P} \in \mathfrak{P}$ for d dimensions $\mathbb{D}_1, \ldots, \mathbb{D}_d$ is a (l, d)-matrix with*

- $\forall i, j, k, l$ *with* $i \neq k$ *or* $j \neq l : \mathcal{P}_{ij} \neq \mathcal{P}_{kl}$
- $\forall \mathcal{P}_{ij} : \mathcal{P}_{ij} = 2^k \quad$ *with* $\quad 0 \leq k \leq d * l - 1$

With these preliminaries, we now can define the address calculation based on bit permutation properly. Multiplying the i^{th} row of X with the i^{th} column of the permutation matrix will result in the contribution of dimension i to the address value. Formally, we have to sum up the values along the diagonal of the result of the matrix multiplication of X and $\mathcal{P}$.

Definition 10.2 (address calculation based on bit permutation)
The address calculation based on a bit permutation $\mathcal{P} \in \mathfrak{P}$ is defined as:

$$\alpha : \Omega \times \mathfrak{P} \mapsto \mathbb{N}_0$$

$$\alpha(x, \mathcal{P}) = \alpha^{\mathcal{P}}(x) = \sum_{i=1}^{d} (X\mathcal{P})_{ii}$$

[1] $j - 1$ stems from the fact that we start counting bits at 0, but dimensions at 1.

To allow for dimensions with different lengths, i.e., different $|\mathbb{D}_i|$, one extends the matrices X and $\mathcal{P}$ to $(d, \max(|\mathbb{D}_i|))$ and $(\max(|\mathbb{D}_i|), d)$ respectively. The bits not used by the corresponding dimension are filled with 0 bits.

Example 10.1: Bit permutation

Bit-interleaving of 3 dimensions with $l = 2$, i.e., 2 bits in binary representation, starting with the first dimension, i.e., dimension 1 provides the highest bit in the address, results in the permutation matrix $\mathcal{P}^{\mathcal{Z}} = \begin{pmatrix} 2^2 & 2^1 & 2^0 \\ 2^5 & 2^4 & 2^3 \end{pmatrix}$.

For a composite key in the order of the dimensions, we get the permutation matrix $\mathcal{P}^{\mathcal{C}} = \begin{pmatrix} 2^4 & 2^2 & 2^0 \\ 2^5 & 2^3 & 2^1 \end{pmatrix}$.

Given the points

$$x_1 = \begin{pmatrix} 2 \\ 2 \\ 3 \end{pmatrix} \longrightarrow X_1 = \begin{pmatrix} 0 & 1 \\ 0 & 1 \\ 1 & 1 \end{pmatrix} \quad \text{and}$$
$$x_2 = \begin{pmatrix} 3 \\ 1 \\ 3 \end{pmatrix} \longrightarrow X_2 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix},$$

we get the following addresses:

$$\alpha^{\mathcal{P}^{\mathcal{Z}}}(x_1) = 0 * 2^2 + 1 * 2^5 + 0 * 2^1 + 1 * 2^4 + 1 * 2^0 + 1 * 2^3 = 57$$
$$\alpha^{\mathcal{P}^{\mathcal{Z}}}(x_2) = 1 * 2^2 + 1 * 2^5 + 1 * 2^1 + 0 * 2^4 + 1 * 2^0 + 1 * 2^3 = 47$$
$$\alpha^{\mathcal{P}^{\mathcal{C}}}(x_1) = 0 * 2^2 + 1 * 2^5 + 1 * 2^1 + 0 * 2^4 + 1 * 2^0 + 1 * 2^3 = 51$$
$$\alpha^{\mathcal{P}^{\mathcal{C}}}(x_2) = 1 * 2^2 + 1 * 2^5 + 1 * 2^1 + 1 * 2^4 + 1 * 2^0 + 0 * 2^3 = 55$$

This example shows the influence of the bit-permutation on the address order and with that on the clustering: $\alpha^{\mathcal{P}^{\mathcal{Z}}}(x_1) > \alpha^{\mathcal{P}^{\mathcal{Z}}}(x_2)$ but $\alpha^{\mathcal{P}^{\mathcal{C}}}(x_1) < \alpha^{\mathcal{P}^{\mathcal{C}}}(x_2)$.

◇

The goal of the address calculation is to provide a mapping of the d-dimensional space to 1-dimensional space that preserves spatial proximity as good as possible. In the following, we restrict our analysis to dimension-order preserving address functions, i.e., bit permutations that preserve the bit order within each dimension.

Definition 10.3 (dimension-order preserving address calculation)
An address calculation $\alpha^{\mathcal{P}} : \Omega \times \mathfrak{P} \mapsto \mathbb{N}_0$ *is called dimension-order preserving, iff* $\forall i, j, 1 \leq i \leq (l-1) : \mathcal{P}_{ij} < \mathcal{P}_{(i+1)j}$

We use $\mathfrak{A}$ to denote the set of all dimension-order preserving address calculation functions. As each $\alpha \in \mathfrak{A}$ preserves the bit order in each dimension, we omit the indices for the bit positions in the future, e.g., instead of $a_{l-1}a_{l-2}b_{l-1}\ldots a_0$ we write $aab\ldots a$.

Without formal proof, we state the following lemma.

Lemma 10.1
For each address calculation $\alpha \in \mathfrak{A}$, the inverse function α^{-1} is a space filling function defining a monotonic ordering on Ω.

The previous example already introduced the permutation matrices $\mathcal{P}^{\mathcal{Z}}$ and $\mathcal{P}^{\mathcal{C}}$ for the Z-order and the C-order resp. for a special case. We can now provide the general definition for both classes of address functions.

Definition 10.4 (C-address)
*An address calculation $C \in \mathfrak{C} = \{\alpha^{\mathcal{P}} \in \mathfrak{A} | \forall i, j : \mathcal{P}_{(i+1)j} = 2 * \mathcal{P}_{ij}\}$ is called **composite address** or short **C-address**.*

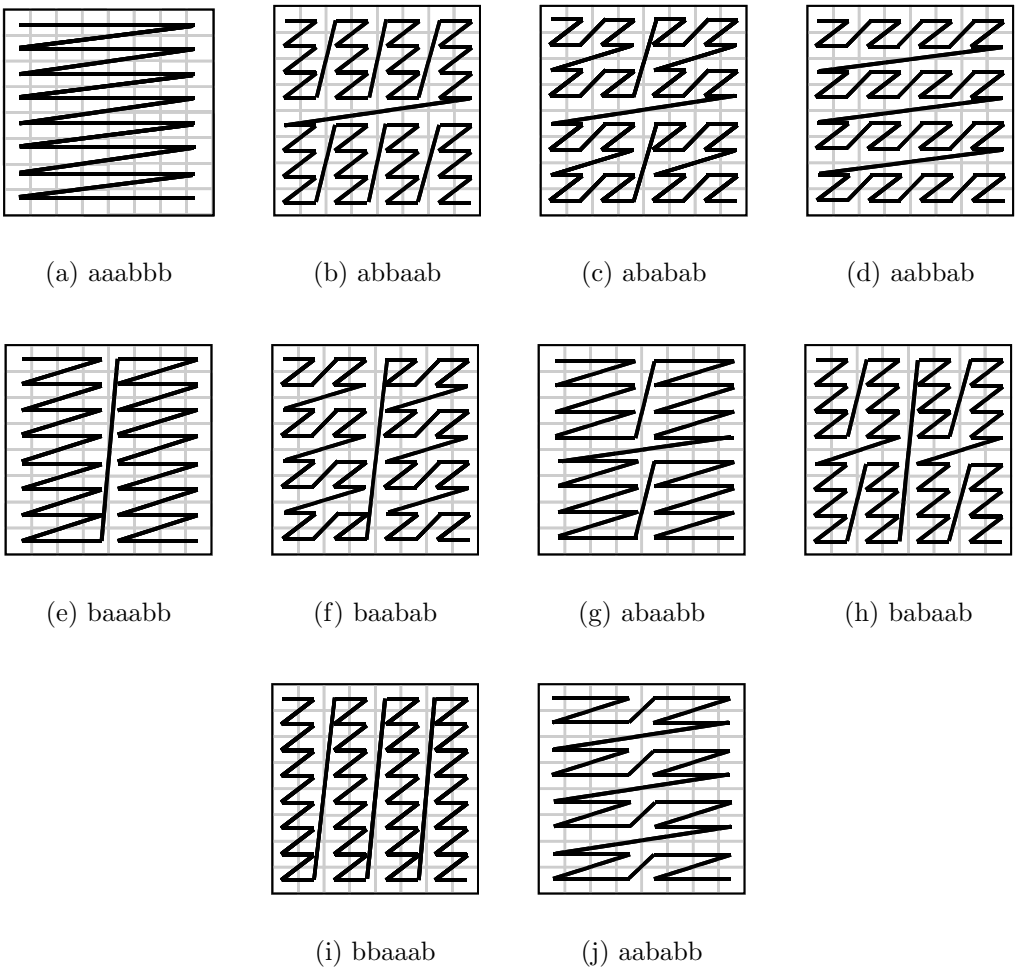
Corresponding to this definition, the binary representation of the composite lexicographic order of two dimensions a and b, each of length 3, would be either *aaabbb* or *bbbaaa*.

Definition 10.5 (Z-address)
An address calculation $Z \in \mathfrak{Z} = \{\alpha^{\mathcal{P}} \in \mathfrak{A} | \forall i, j : \mathcal{P}_{(i+1)j} = 2^d * \mathcal{P}_{ij}\}$ is called Z-address.

Analogously to the C-address, the binary representation of a two-dimensional Z-address with length 3 is either *ababab* or *bababa*.

Example 10.2: Curves of bit permutations

In this example, we show 10 out of 20 possible permutations for a two-dimensional 8x8 universe, i.e., of length 3. Figure 10.1 shows 10 permutations; the other 10 result from switching dimensions a and b. In the graphics, the origin is in the upper left corner and dimension a is oriented vertically and dimension b horizontally.



(a) aaabbb (b) abbaab (c) abaabb (d) aabbab

(e) baaabb (f) baabab (g) abaabb (h) babaab

(i) bbaaab (j) aababb

Figure 10.1: Permutations of an 8x8 universe

◇

10.1.2 Query processing for weighted dimensions

With the new address calculation scheme we can express different weighting of dimensions. Besides the clustering scheme, we also require efficient query processing algorithms for it. It is obvious that there is no difference for point access compared to the standard UB-Tree algorithm.

For the range query processing it is not clear that the NJI (see Section 6.2.1) and NJO (see Section 7.1) algorithms also apply to weighted dimensions. Analyzing the two algorithms more closely one observes that the key requirement is the monotonicity of the addresses, i.e., $\forall x, y \in \Omega : x \lhd y \Rightarrow \alpha(x) < \alpha(y)$. According to Lemma 10.1, this holds for the set $\mathfrak{A}$ of dimension-order preserving address calculations. Consequently, the NJI and NJO algorithm can also be used for weighted dimensions. The modification required is to enhance the function that returns for a given bit position of the address the corresponding dimension, i.e.,

$$dim : \mathfrak{A} \times \mathbb{N} \rightarrow \mathbb{N}_0$$
$$dim^{\alpha}(p) = i \text{ if } \exists k : \mathcal{P}_{ki} = 2^p$$

We write $dim(p)$ for $dim^{\alpha}(p)$ if α is clear from the context.

10.1.3 Specification and notation of weighting

While an address calculation $\alpha \in \mathfrak{A}$ is uniquely specified by the permutation matrix $\mathcal{P}$, this way of defining the weighting is neither very readable nor convenient to specify for the user. An alternative is to specify the binary representation but this leads to a long bit string for high dimensionality and large domains. We therefore need an easier way of specifying the bit permutation.

We concentrate on bit permutations that are combinations of C-order and Z-order. Thus, we use the following simplified notation (in BNF), which also allows to specify any complex permutations:

$$\begin{aligned}
\textit{<indexSpec>} &::= \textit{<indexName : word><weightingSpec>} \\
\textit{<weightingSpec>} &::= (\textit{<weighting>}\ \{,\textit{<weighting>}\}^*) \\
\textit{<weighting>} &::= C|Z(\textit{<attrSpec>}\ \{,\textit{<attrSpec>}\}^*) \\
\textit{<attrSpec>} &::= \textit{<attrName : word>}\ \{(\textit{<bits : integer>})\}
\end{aligned}$$

The total permutation can be expressed by a concatenation of the two weighting schemes: C and Z. C denotes composite order of the specified attributes and Z denotes bit-interleaving of the attribute contributions. The number of bits contributed by an attribute is specified in brackets after the attribute name. If no number is specified, all (still available) bits are contributed. The order of the attribute names in the *<weighting>* clause specifies the composite order or the order of the bit-interleaving, resp.

Example 10.3: Weighting notation

This example introduces the weighting notation.

UB($Z(A, B)$) specifies a standard two-dimensional UB-Tree, i.e., the bit-permutation $ababab \ldots ab$.

UB($C(A, B)$) specifies a standard composite B-Tree, i.e., the bit-permutation $a \ldots ab \ldots b$.

UB$(C(A(5), B(7)), Z(A, B))$ denotes the permutation
$\underbrace{aaaaa}_{5}\underbrace{bbbbbbb}_{7}\underbrace{abab\ldots}_{\text{remaining bits}}$, i.e., the prefix consists of a composite order of 5 A bits followed by 7 B bits; after that the remaining bits are bit-interleaved starting with A.

◇

10.2 Weighting and space partitioning

Up to now, we have only introduced a general addressing scheme based on bit-permutations. In this section, we discuss the relationship between the bit-permutation and the achieved weighting of dimensions, i.e., the influence of the ordering on the space partitioning/clustering.

To talk about the space partitioning of a weighted UB-Tree more formally, we define analogously to Definition 6.2, Definition 6.3, and Definition 6.5 the following.

Definition 10.6 (α-region)
An α-region $[x, y]$ is the space covered by the curve specified by the address calculation $\alpha \in \mathfrak{A}$, i.e., $[x, y] = \{z \in \Omega | \alpha(x) \leq z \leq \alpha(y)\}$.

In Lemma 6.1 we have stated the important connection property of Z-regions, i.e., the property that a Z-region partitions into at most two spatially disconnected sets of points, independently of the dimensionality of the space. We show below that this important property also holds for α-regions.

Definition 10.7 (spatial connection)
*We call two subspaces $Q, P \in \Omega$ **spatially connected** if there exists a pair of points $x \in Q$ and $y \in P$ that only differ in one dimension i and x_i and y_i are $<$-neighbors.*

Theorem 10.1 (connection theorem for α-regions)
Any α-region consists of at most two spatially disconnected sets of points.

The proof of Theorem 10.1 is analogous to the proof for the Z-region connection provided in [Mar99].

Proof 10.1 (connection theorem)
For the proof of the theorem we consider the region $R = [x, z]$. If R consists of disconnected sets of points, we can express R by a set of n α-intervals, each representing a set of connected points, i.e., $R = [x, z] = [x, y_1] \cup [y_1 + 1, y_2] \cup \ldots \cup [y_{n-1} + 1, z]$.

We now assume that $R = [x, z]$ consists of three (pairwise) disconnected sets corresponding to the intervals $[x^s, x^e], [y^s, y^e]$, and $[z^s, z^e]$, with $x^s = x$, $y^s = x^e + 1$, $z^s = y^e + 1$, and $z^e = z$.

$$\begin{aligned} y^s = x^e + 1 \quad \Rightarrow \quad & \exists k \text{ with } x^e_{\ldots(k+1)} = y^s_{\ldots(k+1)},\ x^e_k = 0,\ y^s_k = 1, \\ & x^e_{(k-1)\ldots 0} = 1 \ldots 1,\ y^s_{(k-1)\ldots 0} = 0 \ldots 0 \end{aligned}$$

analogously

$$\begin{aligned} z^s = y^e + 1 \quad \Rightarrow \quad & \exists l \text{ with } y^e_{\ldots(l+1)} = z^s_{\ldots(l+1)},\ y^e_l = 0,\ z^s_l = 1, \\ & y^e_{(l-1)\ldots 0} = 1 \ldots 1,\ z^s_{(l-1)\ldots 0} = 0 \ldots 0 \end{aligned}$$

We distinguish two cases:

1. Case: $y^s_{\ldots(k+1)} = z^s_{\ldots(k+1)}$*, i.e.,* y^s *and* z^s *have the same prefix up to bit position* $(k+1)$
As $y^s < z^s \quad \Rightarrow \quad \exists$ *bit position* $p < k$ *with* $y^s_p = 0$ *and* $z^s_p = 1$.
Let i *be the dimension corresponding to bit position* p*, i.e.,* $i = dim^\alpha(p)$.
Now, we decrement dimension i *of point* $t = (t_1, \ldots, t_d) = \alpha^{-1}(z^s)$ *by one, i.e.,* $t' = (t_1, \ldots, t_{i-1}, t_i - 1, t_{i+1}, \ldots, t_d)$*. Consequently,* t *and* t' *are spatially connected according to Definition 10.7.*

$$\begin{aligned} \alpha(t') < \alpha(t) = z^s \qquad & (\textit{monotonicity of } \alpha) \textit{ and} \\ \alpha(t') \geq y^s \qquad & (\textit{the decrement resets no bit in } z^s \\ & \textit{at a position larger than } p) \\ \Rightarrow \alpha(t') \in [y^s, y^e] & \end{aligned}$$

$\longrightarrow$ *contradiction to* $[y^s, y^e]$, $[z^s, z^e]$ *are spatially disconnected*

2. Case: $y^s_{\ldots(k+1)} \neq z^s_{\ldots(k+1)}$

$$\begin{aligned} y^s_{\ldots(k+1)} \neq z^s_{\ldots(k+1)} \quad & \Rightarrow \quad z^s_{\ldots(k+1)} > y^s_{\ldots(k+1)} \text{ as } z^s > y^s \\ & \Rightarrow \quad y^e > y^s_{\ldots(k+1)} 1 \ldots 1 \\ & \Rightarrow \quad \forall v \text{ with } v_{\ldots(k)} = y^s_{\ldots(k+1)} 1 :\ v \in [y^s, y^e] \end{aligned}$$

Now, we increment dimension $i = dim^\alpha(k)$ *of point* $t = (t_1, \ldots, t_d) = \alpha^{-1}(x^e)$ *by one, i.e.,* $t' = (t_1, \ldots, t_{i-1}, t_i + 1, t_{i+1}, \ldots, t_d)$*. Consequently,* t *and* t' *are spatially connected according to Definition 10.7.*

$$\begin{aligned} \alpha(t') > \alpha(t) = x^e \qquad & (\textit{monotonicity of } \alpha) \textit{ and} \\ \alpha(t')_{\ldots k} = y^s_{\ldots(k+1)} 1 & \\ \Rightarrow \alpha(t') \in [y^s, y^e] & \end{aligned}$$

$\longrightarrow$ *contradiction to* $[x^s, x^e]$, $[y^s, y^e]$ *are spatially disconnected*

□

Definition 10.8 (α-region partitioning)
An α-region partitioning of the universe Ω is a set of α-regions A for which holds:

1. $\forall \alpha_1, \alpha_2 \in A, \alpha_1 \neq \alpha_2 : \alpha_1 \cap \alpha_2 = \emptyset$
2. $\bigcup_{\alpha \in A} \alpha = \Omega$

As two subsequent regions of a partitioning meet at the end and start of the intervals, i.e., $\forall \alpha_1 = [x_1, y_1], \alpha_2 = [x_2, y_2] \in A$ with $y_1 < x_2 \wedge \nexists \alpha = [a, b]$ with $y_1 < a < x_2 : x_2 = y_1 + 1$. We can therefore use either the start or the end values to uniquely identify a region and we call y the *(region) address* of an α-region $[x, y]$.

Definition 10.9 (weighted UB-Tree)
*A **weighted UB-Tree** is any B-Tree using $\alpha \in \mathfrak{A}$ as address function.*

As for the standard UB-Tree, the pagination of the index defines the space partitioning of a weighted UB-Tree.

If we want to talk about the influence of the weighting on the space partitioning, we need some way of characterizing the space partitioning. The problem is that the space partitioning depends on the data distribution making it hard/impossible to capture this formally without making too restrictive assumptions about the data distribution.

There are various factors that influence the space partitioning:

1. Address calculation, i.e., the ordering of the tuples
2. Data size, i.e., the number of tuples
3. Data distribution
4. Page capacity
5. Order of insertion

Starting from the unrealistic assumption of uniform data distribution we will characterize the influence of the first two parameters. The data distribution basically distorts the effects of the first two: the more non-uniform the data distribution the less influence have the other parameters. Further, we assume the same page capacity and neglect the influence of the insertion order as this is hard to control in practice (except for bulk loading etc.).

10.2.1 Symmetry, clustering, and continuity of an ordering

The address calculation defined in this chapter maps the multidimensional domain bijectively to one-dimensional domain. This dimensionality reduction causes some loss of information, namely the neighborhood relationship of points. In the multi-dimensional universe the distance to each direct neighbor in every dimension is 1. Depending on the ordering of the address calculation this may increase significantly

in the one dimensional domain. The concept of the *average neighbor distance* allows for specifying this behavior more precisely.

Definition 10.10 (average neighbor distance of a point)
The average neighbor distance for a point $x \in \Omega$ for an address calculation α with respect to dimension i is:

$$nd_i^\alpha(x) = \begin{cases} \alpha((x_1,...,(x_i+1),...,x_d)) - \alpha((x_1,...,x_i,...,x_d)), & x_i = min^{\mathbb{D}_i} \\ \frac{\alpha((x_1,...,(x_i+1),...,x_d)) - \alpha((x_1,...,(x_i-1),...,x_d))}{2}, & min^{\mathbb{D}_i} < x_i < max^{\mathbb{D}_i} \\ \alpha((x_1,...,(x_i),...,x_d)) - \alpha((x_1,...,(x_i-1),...,x_d)), & x_i = max^{\mathbb{D}_i} \end{cases}$$

To get a measure of the neighborhood preservation for one domain we use the *cumulated neighbor distance.*

Definition 10.11 (cumulated neighbor distance for one dimension)
The cumulated neighbor distance for dimension i with respect to address calculation α is

$$nd^\alpha(i) = \sum_{x \in \Omega} nd_i(x)$$

If the used address calculation is clear from the context, we write nd for nd^α. Without formal proof we state that for a given address calculation all dimensions have different weights.

Lemma 10.2
For all $\alpha \in \mathfrak{A}$:

$$\forall i, j, 1 \leq i, j \leq d, i \neq j : nd^\alpha(i) \neq nd^\alpha(j)$$

Still, we can define a measure of the symmetry of an address function. An address function is called symmetric, if all dimensions have almost the same cumulated neighbor distance. For the following definition we use the *standard deviation (stdv)* of a set of values.

Definition 10.12 (degree of symmetry of an address calculation)
*The **symmetry** of an address calculation $\alpha : \Omega \rightarrow \mathbb{N}_0$ is calculated as:*

$$symmetry(\alpha) = -stdv(\{nd(i) | 1 \leq i \leq d\})$$

As the next example shows, there are bit permutations that are more symmetric than bit interleaving, i.e., the Z-order. However, we will show in the following sections, that the Z-curve has the best scalability behavior, making it more suitable for dynamic applications.

Example 10.4: Symmetry and clustering

Lets again consider a two-dimensional 8x8 universe, allowing for 20 permutations.

Table 10.1 presents the cumulated neighbor distance and the symmetry for all 20 permutations.

Table 10.1: Symmetry of all 20 permutations

Pattern	nd(a)	nd(b)	Symmetry
aaabbb	512	64	-316,78
aababb	448	96	-248,9
aabbab	416	112	-214,96
aabbba	400	128	-192,33
abaabb	384	160	-158,39
ababab	352	176	-124,45
ababba	336	192	-101,82
abbaab	320	208	-79,2
abbaba	304	224	-56,57
abbbaa	288	256	-22,63
baaabb	256	288	-22,63
baabab	224	304	-56,57
baabba	208	320	-79,2
babaab	192	336	-101,82
bababa	176	352	-124,45
babbaa	160	384	-158,39
bbaaab	128	400	-192,33
bbaaba	112	416	-214,96
bbabaa	96	448	-248,9
bbbaaa	64	512	-316,78

◇

Neighbor distance and weight of a dimension

It is still open how we can express the weight of a dimension, reflecting that higher weight means more preference. In our context, preference means better clustering. Recalling Definition 2.14, clustering means to place tuples that one is interested in close together. Translating this to the ordering function, this means, that points that have the same value in one dimension should be close together, independently of the values of the other dimensions. Using the neighbor distance as a metric leads to counter-intuitive results: the permutation *aaabbb* clearly favors dimension a, but the neighbors in a are farther away than in dimension b, i.e., $nd(a) > nd(b)$ (see Table 10.1). The problem of the neighbor distance is that it does not capture the semantics of clustering: points with the same value x_i in dimension i but different

values in the other dimensions are not considered to be neighbors of point x w.r. to dimension i.

To get a better grip on the relationship of the ordering and the clustering we therefore define the *clustering degree* as follows. We thereby take into account that all points in the multidimensional space that have the same value in one dimension should be close together in one-dimensional space, if we want to cluster according to this dimension.

Definition 10.13 (clustering degree of a dimension)
*The **clustering degree** of an address calculation $\alpha \in \mathfrak{A}$ with respect to dimension i is the average length of the interval to which all multidimensional points with the same value in dimension i are mapped to, i.e.,*

$$cd_i^\alpha = \frac{1}{|\mathbb{D}_i|} * \sum_{j=min^{\mathbb{D}_i}}^{max^{\mathbb{D}_i}} (max\{\alpha(x)|x_i = j\} - min\{\alpha(x)|x_i = j\} + 1)$$

With this definition, we can state the subsequent lemma.

Lemma 10.3 (Weight of a dimension)
A dimension i has more weight than dimension j in the address calculation α if dimension i has a smaller clustering degree, i.e., $cd_i^\alpha < cd_j^\alpha$.

Example 10.5: Weighting and clustering

For our standard two-dimensional 8x8 example, Table 10.2 depicts the clustering degree for all 20 permutations.

◇

10.2.2 Resolution: significant address prefix

So far, we have considered the complete address, i.e., we have assumed the highest possible resolution of one point. In an index, multiple tuples are stored on one page. Each page corresponds to one α-region and consequently stores tuples within one interval.

Assuming a partitioning of the universe into P pages, one minimally needs $l = \lceil \log_2 P \rceil$ bits for uniquely addressing all pages, i.e., regions. Thus we focus on the significant header of length l of the region address, called the *prefix*. For perfect uniform data distribution the significant prefix has exactly length l, getting longer and longer for more non-uniform data distributions.

As the pages constitute the smallest addressable unit on secondary storage we can limit our weighting analysis to the prefix of the address. Each bit in the prefix causes a split in the middle of the corresponding dimension. Each further bit recursively causes further splits.

Table 10.2: Clustering degree of different permutations

Pattern	cd_a	cd_b
aaabbb	9,14	65,14
aababb	13,71	60,57
aabbab	16,00	58,29
aabbba	17,14	57,14
abaabb	22,86	51,43
ababab	25,14	49,14
ababba	26,29	48,00
abbaab	29,71	44,57
abbaba	30,86	43,43
abbbaa	33,14	41,14
baaabb	41,14	33,14
baabab	43,43	30,86
baabba	44,57	29,71
babaab	48,00	26,29
bababa	49,14	25,14
babbaa	51,43	22,86
bbaaab	57,14	17,14
bbaaba	58,29	16,00
bbabaa	60,57	13,71
bbbaaa	65,14	9,14

Definition 10.14 (Contribution)
Given an address calculation $\alpha \in \mathfrak{A}$. The ***contribution*** *of dimension i, contrib(i), to the prefix of length l, is the number of bits in the prefix belonging to dimension i. A contribution of contrib(i) leads to $2^{contrib(i)}$ divisions/splits in dimension i.*

The finer a dimension is split, the more precisely a range restriction in the query can be approximated and the less regions have to be accessed. The number of bits per dimension in the prefix is therefore an important metric.

The relative weighting of two dimensions i and j is expressed by the fraction of the divisions in each dimension, i.e., by $\frac{2^{contrib(i)}}{2^{contrib(j)}}$.

Example 10.6: Space partitioning for weighted addresses

With this example, we illustrate the influence of the address calculation on the space partitioning. Consider a two-dimensional 8x8 universe. We consider two address functions:

- Z-curve: ababab
- C-curve: aaabbb

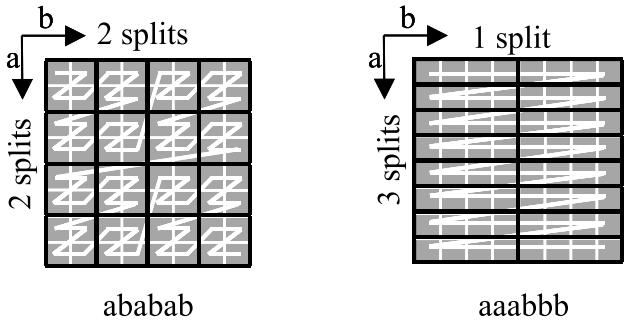
Assuming a page capacity of four, we require 16 pages to store the complete universe. For 16 pages, a prefix of $\log_2 16 = 4$ bits of the address

suffices to identify the pages. Table 10.3 presents the contribution and resolution of the two dimensions for both curves.

Table 10.3: Contribution

Curve	$contrib(a)$	$contrib(b)$
Z	2	2
C	3	1

Figure 10.2 shows the space partitioning of the Z-curve (Figure 10.2(a)) and the C-curve (Figure 10.2(b)).



(a) Z-curve (b) C-curve

Figure 10.2: Space partitioning of different bit permutations

◇

For analyzing the role of the bit order in the prefix, we need one more concept. Without formalization, we introduce the notion of *jumps* of a space-filling curve. A jump is a position on the curve where it is not continuous. Jumps on a space-filling curve defined by a bit permutation are caused by changes in the binary contribution, i.e., between bits of different dimensions. The lower the bit position, the more but smaller jumps are introduced. Figure 10.3 illustrates this for the Z-curve.

The contribution is the key influence factor for the weight of a dimension. Still, the ordering may further improve the weighting for one dimension. Non-uniform data distributions lead to irregular space partitioning causing varying prefix lengths of the regions. For such cases, one can further optimize the partitioning for one dimension in such a way that it is less influenced by the position of the query box in that dimension. Placing the bits of one dimension at the end of the prefix prevents larger jumps in this dimension (cf. permutations $aab*, aba*, baa*$ in Figure 10.1). However, this optimization is only useful if there is no change in the prefix length, i.e., the data volume and the data distribution is static.

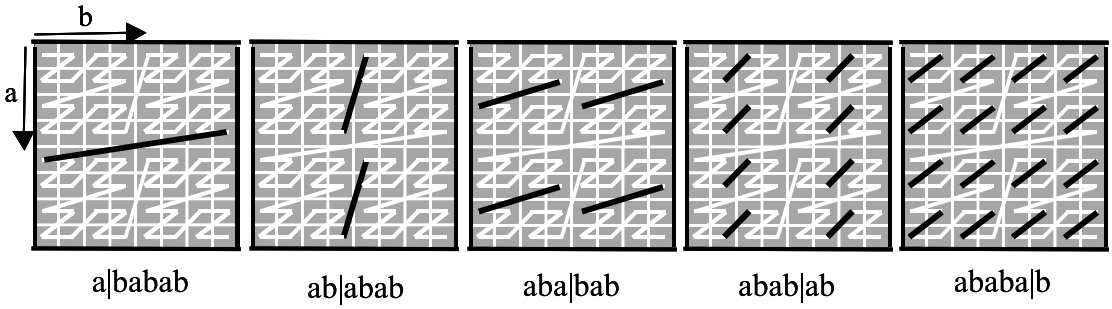


Figure 10.3: Relationship between bit pattern and jumps/discontinuity of a space-filling curve

10.2.3 Scalability and robustness

With the concepts of contributions and jumps we can now define exactly the weighting of the dimensions for a given/targeted data base size. As we do not only consider static databases we have to cope with growing data sizes, i.e., with increasing prefix lengths. If the prefix grows, the contribution of the dimension changes according to the bit pattern of the permutation. To maintain the relative weighting of dimensions with growing prefixes, the postfix has to contribute iteratively one bit from each dimension. This means, in the postfix the remaining bits of all dimensions are bit-interleaved. The following example illustrates the behavior of different permutations.

Example 10.7: Scalability

Figure 10.4 illustrates the change of the relative weighting for four indexes on a two-dimensional $2^{10} \times 2^{10}$ universe:

- *W1(Z(a,b))*: Standard UB-Tree starting interleaving with dimension a
- *W2(C(a,b))*: Standard composite B-Tree with order a, b
- *W3(C(b(4),a(10)),C(b))*: four bits of b, followed by all a bits and then the remaining six bits of b
- *W4(C(b(4),a(4)),Z(a,b))*: 4 bits of b, followed by 4 bits of a, then interleaving the remaining bits of both dimensions starting with a

Now consider the following four indexes with a weighting of $a : b = 4 : 1$.

1. *V1(C(a(3),b(1)),Z(a,b))*: robust weighting due to interleaving after target prefix
2. *V2(C(b(1),a(10)),C(b))*: optimization of a for target size
3. *V3(C(b(1),a(3)),Z(a,b))*: as *V1* but with different prefix order
4. *V4(C(a(3),b(1)),Z(b,a))*: as *V1* but with different interleaving in the postfix

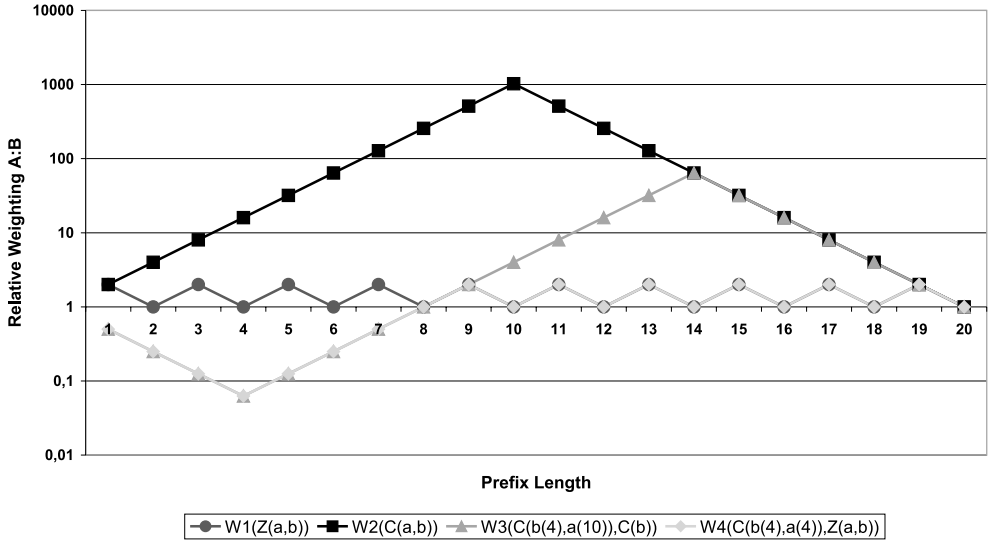


Figure 10.4: Change of weighting with growing database size

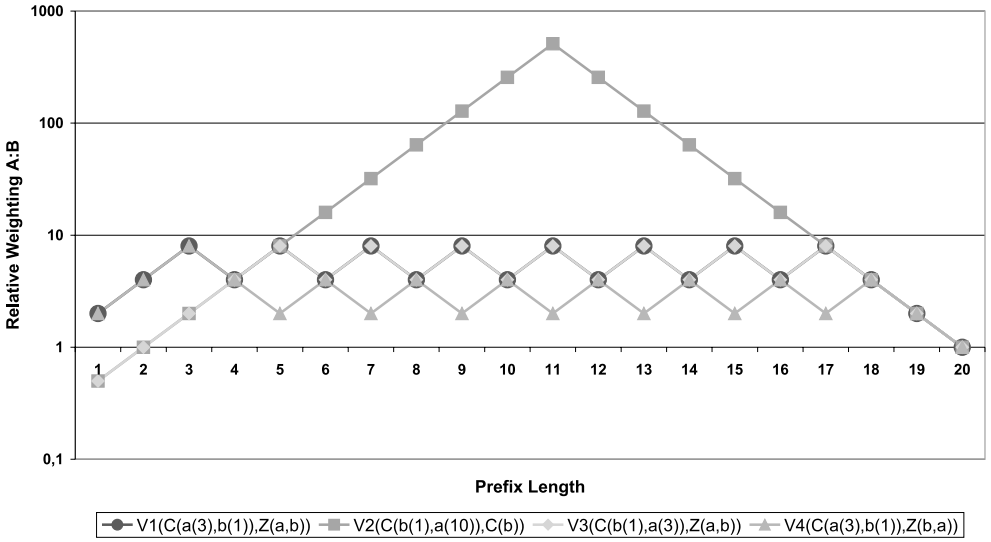


Figure 10.5: Scalability of Weighting

Figure 10.5 shows the behavior of those indexes for growing database size. The first observation is that the targeted weighting cannot be achieved for databases smaller than 2^4 pages, as we require at least 3 bits from a and 1 bit from b for the relative weighting of 4:1. For larger prefixes, the indexes with bit-interleaved postfix are able to maintain the target weighting (at least every second bit). Index *V2* has been optimized for a fixed database size and consequently favors dimension a with growing prefix until all a bits are exhausted. From that point on the weight of dimension b increases to eventually reach a 1:1 weighting.

◇

10.3 Weighted dimensions in practice

Finally, we present some performance results of different weighting schemes for our GfK3D (see Section 4.3) data set, for one year , i.e., ≈ 14 million tuples. With a page capacity of 59 tuples and an average page utilization of 75% this results in a total of $P = 312394$ pages. Thus, we have a minimal prefix of length $l = \lceil \log_2 P \rceil = 19$. Table 10.4 provides the detailed break-down of the key length of the three dimensions Product (Pr), Segment (Se), and Time (Ti); the dimension keys are HC keys, resulting from encoding of the dimension hierarchies.

Table 10.4: Address length analysis for GfK3D

Dimension	**Hierarchy level**	**Fan-out**	**Bit length (level)**	**Bit length (total)**
Time	Year	4	2	5
	4_Month_Period	3	2	
	2_Month_Period	2	1	
Product	Sector	14	4	29
	Category	9	4	
	ProductGroup	83	7	
	Item	15601	14	
Segment	Country	16	4	24
	Region	19	5	
	Micromarket	6	3	
	Outlet	2202	12	

Query and weighting specification

Before we present the chosen weighting of the dimensions, we have to take one more look on the queries we are going to measure. We use the following query sets representing roll-ups in the Product dimension:

- Product-Group (PG) series: total sales of one product group in a two-months period for a specific country (604 queries)
- Category (CAT) series: total sales of one category in a two-months period for a specific country (30 queries)
- All products (ALL) series: total sales in a two-months period for a specific country, i.e., no restriction in the Product dimension (16 queries)

Summarizing the query specification, we observe that the Time dimension is always restricted on the lowest hierarchy level, the Segment dimension is always restricted on the highest hierarchy level, and the restriction in the Product dimension varies.

Optimizing the weighting of the UB-Tree for this special workload, we derive the following prefix contributions:

- Time: 5 bits covering the complete dimension

- Segment: 4 bits; sufficient to cover the first hierarchy level
- Product: 10 bits, covering the first two hierarchy levels and parts of the Product-Group level

We compare four indexes to evaluate the influence of weighting:

- **UB**: standard UB-Tree, i.e, *UB(Z(Ti,Pr,Se))*
- **W1**: weighted UB-Tree using bit-interleaving in the prefix and in the postfix, i.e., *W1(Z(Se(4),Pr(10),Ti(5)),Z(Se,Pr))*
- **W2**: same weighting as W1, but composite order in the prefix to optimize further for dimension P, i.e., *W2(C(Se(4),Ti(5),Pr(29)),C(Se))*
- **W3**: same weighting as W1, but dimension order in the prefix according to the weighting, i.e., $W3(\underbrace{C(Pr(2),Ti(1),Se(1)),\ldots,C(Pr(2),Ti(1),Se(1))}_{5\ times},Z(Pr,Se))$

Product-Group series

Figure 10.6 presents a zoom into the 300 smallest queries of the PG series. The results clearly reveal the effects of weighting: the weighted indexes *W1*, *W2*, and *W3* perform better than *UB* due to their weighting. Especially for small result sets the effect is tremendous: *W2* needs more than 25 times less page access than *UB*. As the size of the result set grows, this major difference vanishes.

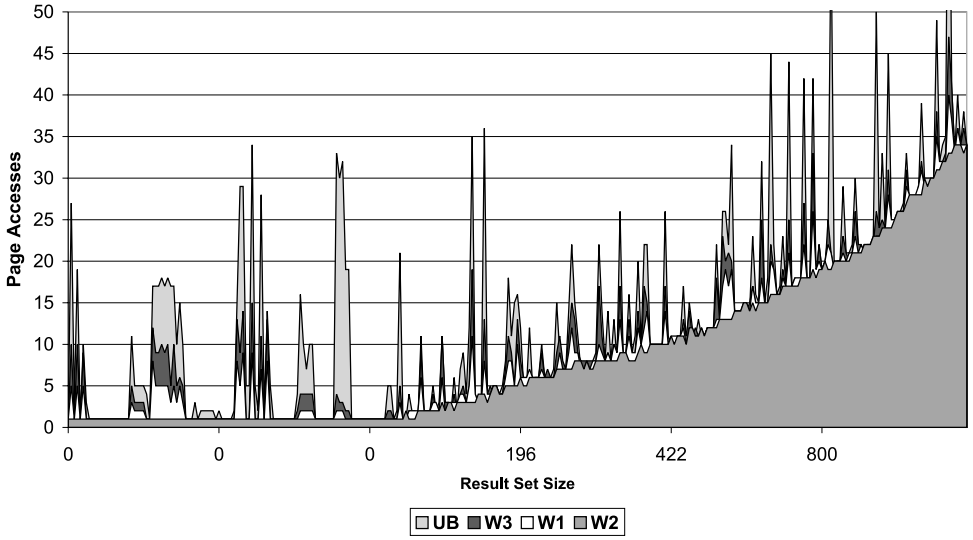


Figure 10.6: Results of PG series

There is also a big difference between *W1*, *W2* and *W3* that have the same dimension weighting. This is an effect of the optimization of the prefix order in

W2 allowing for more continuity w.r. to dimension P. Consequently, *W2* is less susceptible to the location of the query boxes (assuming that queries with similar result set size have similar sized query boxes, but definitely different locations). For the same reason *W1* performs better than *W3*: in the weighting of *W1* the bits for the Product dimension are more to the end of the prefix than in *W3* and thus *W1* is more robust w.r. to the query box location.

Figure 10.7 summarizes the relative performance of *UB*, *W1*, *W2*, and *W3*. The first group of pillars shows the overall performance for all 604 queries of this benchmark in number of page accesses, relative to the standard UB-Tree *UB*. The other two groups only consider the first 300, resp. 150 queries with the smallest result-sets, where the weighted UB-Trees impressively demonstrate their ability to efficiently approximate small query boxes.

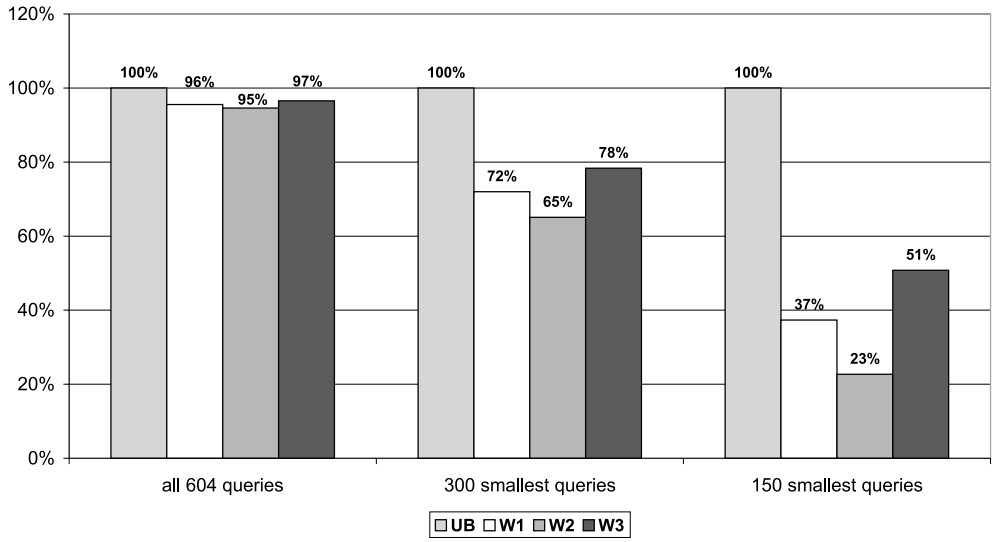


Figure 10.7: Relative performance of weighted UB-Trees

Category and all products series

Finally, we will have a look on the two other GfK benchmark series, that are not especially favored by this weighting (the restriction in the Product dimension is reduced), in order to get a feeling on how robust a specific weighting is in a real-world application.

Figure 10.8 shows the relative performance of the four UB-Trees for both query sets. As expected, the advantage of the weighted UB-Trees diminishes the weaker the restriction in the Product dimension gets. Still, the weighted UB-Trees do not perform worse than the standard UB-Tree as they all contain the restricted bits in the other dimensions.

Obviously, the standard UB-Tree will be favored if the preferences shift further towards the Segment dimension. Queries with stronger restrictions, i.e., restrictions to lower hierarchy levels, in the Segment dimension will be processed faster by the standard UB-Tree. This is depicted in Figure 10.9 that shows the results of 5 drill-

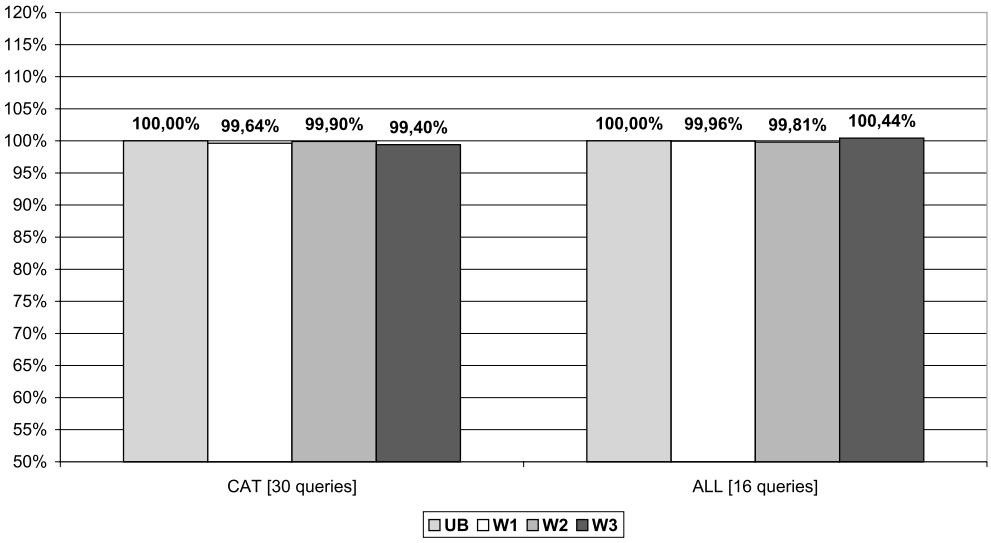


Figure 10.8: Results of CAT and ALL series

down queries to one outlet in the Segment dimension, for one 2-month period and no restriction in the Product dimension. While *W1* and *W3* only show a modest difference compared to *UB* (smaller than a factor of 5), the performance of *W2* degenerates heavily as only the restriction on the Time dimension and the restriction on the first level in the Segment dimension can be utilized.

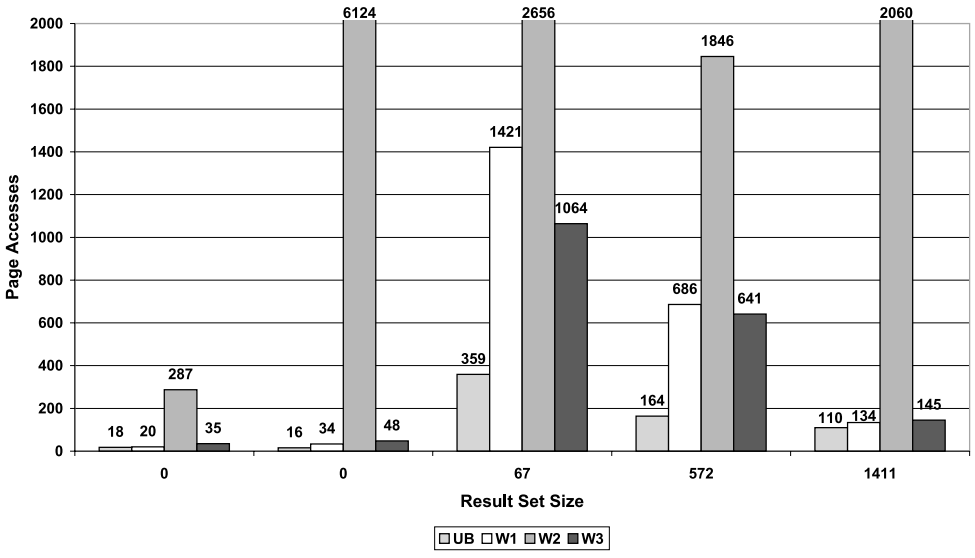


Figure 10.9: Results of drill-down queries to one outlet

10.4 Chapter notes and related work

In this chapter we have introduced an enhanced address calculation scheme for UB-Trees that allows for arbitrary weighting of dimensions. By extending the address calculation from the concept of bit-interleaving to (dimension order preserving) bit-permutation we are able to specify different weighting of dimensions.

We have evaluated the influence of the bit order in the address on the weighting. As expected, a dimension has more weight, the more bits it contributes to the significant prefix of the address. For static database one can further optimize the clustering for one dimension with specific ordering of the prefix bits. However, in dynamic scenarios, which are more common, the goal is to find a robust weighting that is stable with growing database size.

Our benchmarks with the GfK3D data set have confirmed that for graded queries an appropriate weighting of the dimensions can significantly improve the query performance. Especially small queries are improved by factors of 5 and higher.

Still, defining the best weighting for a given application is a non-trivial problem. For the GfK example it is relatively easy as the hierarchy specification provides a good guideline, but this can not be expected for general applications. Further, many users will not be able to specify the weighting of the dimensions as a bit permutation. Therefore, there is the need for a extension of the standard interface, i.e., SQL, that allows for naturally specifying the wanted weighting. Or, as it is done already for index-selection [CN97, CN98b], the decision is left completely to the DBMS, just taking a target workload as input for which the dimension weighting has to be optimized.

Related work

To the best of our knowledge, this is the first work to address the issues of weighted dimensions for indexing. All multidimensional access methods we are aware of either support pure symmetrical handling of dimensions or one specific, non-controllable weighting. With composite B-Trees one can only specify a very strict weighting of attributes.

Chapter 11

The Skipper technique: Processing multi-attribute restrictions on composite indexes

As discussed in the previous chapter, with the concept of weighted dimension we can also specify a composite key order for UB-Trees. The difference between a standard B-Tree with composite key and a correspondingly weighted UB-Tree is the way range queries are processed. The question that we want to address in this chapter is, whether the UB-Tree range query processing works better than the standard range query processing for composite keys.

For our description of the algorithms, we assume a B-Tree with the following operations:

- t=search($index$, k): positions to the first tuple $t \geq k$ in key order and returns it, i.e., it performs a point search on $index$
- getnext($index$): positions to the next tuple from the current position

11.1 The standard range query algorithm on composite keys

Figure 11.1 shows the standard query algorithm (Standard) for handling a multi-attribute restriction with composite key indexes [GR93]: the lower limit ql of the multidimensional interval is used to search for the first tuple in the composite key index, that satisfies the query condition. Then all consecutive tuples in composite key order up to the tuple qh are retrieved. If tuple a satisfies the overall query condition $a \in [[ql, qh]]$, it is returned to the caller for further processing.

Lemma 11.1
The standard algorithm requires a scan over an interval on the composite key values that is defined by the common prefix of ql and qh and the first non-identical component of ql and qh, i.e., formally, for $k \leq d$ and $ql_i = qh_i$ for all $i < k$, the composite

```
1:  KEY a, ql, qh;
2:  a = search(composite-index,ql);
3:  while (a <= qh) {
4:     if(a in [[ql,qh]]) {
5:        output(a);
6:     }
7:     a = getnext(composite-index);
8:  }
```

Figure 11.1: Standard algorithm for range queries on composite keys

key interval $[ql_1 \circ \ldots \circ ql_k \circ \ldots \circ ql_d \ , \ ql_1 \circ \ldots \circ ql_{k-1} \circ qh_k \ldots \circ qh_d]$ *is retrieved and examined by the algorithm.*

Proof 11.1
The Lemma follows directly from the while-loop of the standard algorithm in Figure 11.1 and the definition of the composite ordering.

11.2 The Skipper technique for multi-attribute restrictions

We now introduce the *Skipper technique* or *index skipping*, based on the same principle as the UB-Tree range query algorithm. We map the restriction in form of a multidimensional interval to a set of one-dimensional intervals in composite key order. This is achieved by using the index as shortcut each time when the standard composite key query algorithm investigates a key value not belonging to the result set. Figure 11.2 shows the Skipper technique in the else-part of the algorithm (lines 9-15).

```
1:  KEY a, s;
2:  a = search(composite-index,ql);
3:  // a = min {x | x in composite-index && x >= ql }
4:  while (a <= qh) {
5:     if (a in [[ql, qh]]) {
6:        output(a);
7:        a = getnext(composite-index, a);
8:     }
9:     else { // Skipper
10:       s = nextJumpIn(a, ql, qh);
11:       // s = min {b | b >= a && b in [[ql,qh]]} || s = EOF
12:       if (!EOF)
13:          a = search(composite-index, s);
14:          // a = min {a |a in composite-index && a >= s}
15:    }
16: }
```

Figure 11.2: Skipper algorithm

As we have pointed out before, skipping means to subdivide the interval $[ql_1 \circ \ldots \circ ql_k \circ \ldots \circ ql_d \ , \ ql_1 \circ \ldots \circ ql_{k-1} \circ qh_k \ldots \circ qh_d]$ into a union of subintervals $[l_1, u_1], \ldots, [l_m, u_m]$ that represent the smallest cover of the query box $[[ql, qh]]$. This is achieved dynamically by calculating the next lower sub-interval limit l_{i+1}, after the previous sub-interval upper bound u_i has been exceeded in line 13. Note that the algorithm in Figure 11.2 does not need to store the intervals, thus the variables l_i and u_i do not occur in the code. The variable s in the code stores l_i for the current interval. The next lower bound l_{i+1} is calculated from the current value a, which is larger than u_i (the upper bound of the current interval) when the Skipper-part of the algorithm in Figure 11.2 is entered. Then $s = l_{i+1}$ is calculated as the minimal composite key value larger than a, and contained in the multidimensional interval $[[ql, qh]]$. The *nextJumpIn*-algorithm for this computation is shown in Figure 11.3.

```
1:  KEY nextJumpIn(KEY a, KEY ql, KEY qh) {
2:     int j, i = 1;
3:
4:     while( (a[i] >= ql[i]) && (a[i] <= qh[i]) ) i++;
5:     for(j = i-1; j >= 1 ; j--) {
6:        if(a[j] < qh[j]) break;
7:     }
8:     if ( j == 0)  //we have reached the end of the query box
9:        return EOF;
10:    for(i = 1; i <= j; i++) s[i] = a [i];
11:    s[j]++;
12:    for(i = j+1; i <= d; i++) s[i] = ql[i];
13:    return s;
14: }
```

Figure 11.3: nextJumpIn algorithm: Computing the next lower interval boundary s for $[[ql, qh]]$ and current position a

The while-loop in line 4 determines the first attribute in the composite key that is beyond the upper bound. Starting from this position, the for-loop in lines 5 - 7 finds the last attribute in the prefix examined so far, for which the value can still be advanced within the interval. Once this position has been found, nextJumpIn determines the lower bound of the next interval as follows: all values of the attributes in front of the current position are unchanged (line 10), the value of the attribute at the current position is incremented, i.e., set to the next value in domain order (line 11), and all other attributes are reset to the lower bound of the intervals (line 12). The nextJumpIn algorithm returns EOF if there is no more tuple to return, i.e., if we call nextJumpIn with tuple a with $qh < a$. In an optimized implementation, the $O(d)$ attribute checks can be undertaken on the fly while checking $a \in [[ql, qh]]$, since this requires the same comparisons. Then this linear algorithm for the computation of s has a complexity of $O(d)$ attribute copy operations. Thus the CPU overhead compared to the standard algorithm is merely the d assignments, which only take place once for each interval skip. We call this Skipper technique or index skipping because it navigates through the composite key index and skips key ranges that cannot be part of the result set.

Example 11.1: Index skipping

Let us assume a market research company that tracks the sales of stores in several countries for products in two product categories on a daily basis. The main table of that application is a relation *R(YEAR, COUNTRY, MONTH, CATEGORY, DAY, SHOP, ITEM, SALES)* with a composite key index on the attributes YEAR, COUNTRY, MONTH, and CATEGORY and some other attributes in the given order (this is actually a simplified schema of GfK). The upper part of Figure 11.4 shows a part of the sequential or paginated disk layout of the relation R for the composite key order $<_{\text{YEAR}\circ\text{COUNTRY}\circ\text{MONTH}\circ\text{CATEGORY}\circ\ldots}$, which stores all data for the year 1999 (in our example the company tracked sales data for Hungary and Germany in 1999 for the product categories "Brown Goods" and "White Goods".

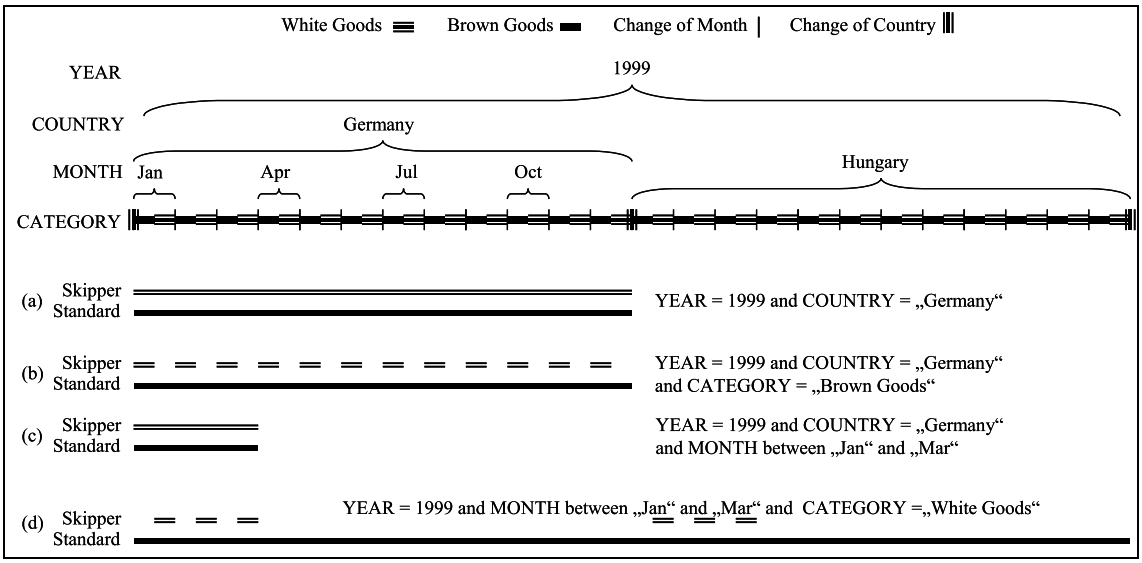


Figure 11.4: Comparison of Skipper and Standard

The lower part of Figure 11.4 shows four queries and the range that both Skipper and Standard retrieve from disk for answering the queries. A restriction to year 1999 in Germany is a restriction of the prefix of the composite key (Figure 11.4(a)), thus both Skipper and Standard retrieve the same range of disk pages from secondary storage. The additional restriction on CATEGORY to "Brown Goods" in Figure 11.4(b) cannot be utilized by the standard algorithm. If the sales of "Brown Goods" and "White goods" in "Germany" in 1999 are roughly the same, then Skipper retrieves 50% less data than Standard for this query and can thus be expected to be around 50% faster than the standard algorithm. Figure 11.4(c) shows a query, where the last restricted attribute of the composite key is restricted to a range to get all sales for Germany in the first quarter of 1999. In this case, the multi-attribute query again transforms into a single interval on the composite key and is therefore

handled identically with Skipper and Standard. In Figure 11.4(d), the COUNTRY attribute, the second leading attribute, is not restricted by the query. Thus the standard algorithm can neither utilize the range restriction on the first quarter of 1999 nor the restriction to "White Goods". If we assume the same amount of data stored for each category in each country on each day, Skipper would only access one eighth of the disk pages that Standard accesses to answer the query.

◇

Example 11.1 is intended to give an intuitive understanding of the Skipper algorithm. It is misleading because of two factors, though.

First, the disk page order is usually not identical to the composite key order. The page clustering is only available for static databases that have been created by mass loading and have not been updated after creation. With clustering indexes, one can expect tuple clustering, i.e., the tuples are clustered on the disk pages with respect to the composite key order, but not between the pages.

Second, the figure suggests that each key range stores the same number of values, since all lines for "Brown Goods" and "White Goods" have the same length. This is not true for practical data sets. However, both the page clustering and the uniformity assumption can be safely dropped without changing the basic picture. First, Standard benefits more from page clustering than Skipper, since the standard algorithm will read at least the pages that skipper reads as well. Second, if the data distribution is not uniform, then the number of accessed useful data pages varies depending on the query. This results in an even better relative performance of Skipper for result sets created by queries with restrictions into a more sparsely populated part of the database. In general, the Skipper relates response time to result set size more closely than the standard algorithm does.

11.3 Pre-computation of intervals

Like for the UB-Tree range query algorithm, we are able to pre-compute the intervals in which a multidimensional query box on a composite key decomposes to.

```
1:  KEY nextJumpOut(KEY a, KEY ql, KEY qh) {
2:     int i=d, j = 1;
3:
4:     while( (ql[i] == MIN[i]) && (qh[i] == MAX[i] && i > 1) ) i--;
5:     for(j = 1; j < i ; j++) s[j]=a[j];
6:     for(j = i; i <= d; j++) s[j] = qh[i];
7:     return s;
8:  }
```

Figure 11.5: nextJumpOut algorithm: computing the next upper interval boundary s for $[[ql, qh]]$ and current position a

The nextJumpOut can be optimized as the computation in the while-loop has only to be performed once for a given query box. Using nextJumpIn of Figure 11.3 and nextJumpOut of Figure 11.5, the composite key interval set $\{[l_1, u_1], \ldots, [l_n, u_n]\}$ corresponding to the multidimensional restriction $[[ql, qh]]$ can be constructed as shown in Figure 11.6.

```
l_1 = ql;
u_i = nextJumpOut(l_i, ql, qh); //i=1, ...,n
l_{i+1} = nextJumpIn(u_i + 1, ql, qh); //i=2, ...,n-1
```

Figure 11.6: Computing the interval set $[l_1, u_1], ..., [l_n, u_n]$ for $[[ql, qh]]$ with nextJumpIn and nextJumpOut

Pre-computation of the interval set allows for a middleware approach, since a query rewriting mechanism on top of a DBMS may form a multi-attribute restriction into the union of subqueries, each of which fully specifies the composite key. However, this approach has several disadvantages over the dynamic approach of Section 11.2, similar to the ones we have discussed in the context of the nextJumpOut algorithm for the UB-Tree (cf. Section 7.1).

First, pre-computation will generate a fairly long query statement, since each interval results in a sub-query specifying a restriction that has at least the length of the original restriction. For n intervals, the size of the rewritten query statement will be at least n times the size of the original query statement. This may cause problems in practice, since the query string length of a DBMS is usually limited.

Second, many DBMS optimizers have problems to optimize queries with long query statements consisting of restrictions which result in unions of disjoint intervals on the same table. During our measurements, we experienced on several commercial DBMSs that the optimizer preferred a full table scan over multi-interval index access on the B-Tree, although the multi-attribute index access would have resulted in far better query response time.

Third, depending on the query and data distribution, some or even many intervals generated by the middleware approach may not occur during the execution of the dynamic Skipper algorithm. While the middleware approach generates all possible composite key intervals for a query box, the dynamic Skipper algorithm adapts to the actual data distribution. The search function after the nextJumpIn calculation in the Skipper part of Figure 11.3 may move the current tuple over some intervals that would be created by the middleware approach. Thus intervals not containing any data are dynamically skipped.

11.4 Integration aspects of Skipper

This section briefly discusses some implementation aspects of the Skipper algorithm.

11.4.1 Integration into a DBMS kernel

As shown in Figure 11.1 and Figure 11.2, the extension of the standard algorithm of processing composite keys to the Skipper technique is straightforward. Especially the possibility of combining the nextJumpIn computation with the post-filtering significantly reduces the overhead of Skipper. A further optimization is to perform the nextJumpIn only once per accessed page, instead of computing the skips each time a tuple outside the query box is found. In the first step, all tuples of a loaded page are post-filtered like in the standard algorithm. If the last tuple on the page is inside the query box, Skipper continues with processing the next page. If the last tuple is outside the query box, Skipper continues with the nextJumpIn computed from this tuple. Thus the small overhead of nextJumpIn occurs only once per page and at the same time multiple jumps to the same page are avoided.

11.4.2 Prototype implementation of the Skipper algorithm

For our performance measurements we have implemented the Skipper algorithm on top of a commercial DBMS. Given a query box $Q = [[ql, qh]]$ of multidimensional restrictions, corresponding to the SQL-query in Figure 11.7, the prototype implementation works as follows.

```
SELECT A_1 ,..., A_d
FROM R
WHERE A_1 BETWEEN ql_1 AND qh_1
AND ...  AND
A_d BETWEEN ql_d AND qh_d
```

Figure 11.7: Original query

We open a cursor with the SQL query given in Figure 11.8, which besides returning the result tuples of Q also retrieves the tuples we need to identify skips, i.e., tuples that are not in the query box.

```
SELECT A_1 ,..., A_d
FROM R
WHERE (A_1>ql_1)
OR (A_1=ql_1 AND A_2>=ql_2)
OR ...  OR ...
OR (A_1=ql_1 AND A_2=ql_2 AND ...  AND A_d>=ql_d)
```

Figure 11.8: Skipper query

The disjunction of the d predicates handles all possible violations of the original query box (see Example 11.2). To detect the possibility of skipping, we have to

post-filter all tuples returned by the cursor and as soon as we find a tuple violating Q, we start skipping. With the result of nextJumpIn for this tuple we generate a new query corresponding to the next interval and continue the processing with a new cursor for this statement. This implementation causes significant overhead to the Skipper processing in terms of CPU (post-filtering is done twice) and I/O (multiple page accesses cannot be prevented). However, it suffices to show the benefits of the Skipper technique w.r. to I/O cost.

Example 11.2: Cursor implementation of Skipper

Let R be a relation with the three-dimensional base space $\Omega = \mathbb{D}^3$, $\mathbb{D} = \{1, 2, 3, 4\}$, and the key order A_1, A_2, A_3. Now consider the query box $Q = [(2, 2, 2), (3, 3, 3)]$.

The first cursor has the following predicate:

$$\begin{array}{ll} A_1 > 2 & \text{OR} \\ (A_1 = 2 \text{ AND } A_2 \geq 2) & \text{OR} \\ (A_1 = 2 \text{ AND } A_2 = 2 \text{ AND } A_3 \geq 2) & \text{OR} \end{array}$$

The behavior of Skipper now depends on the data; we now show how Skipper acts in case of different tuple streams returned by the cursor. We just consider the first tuple t returned that is NOT part of the query box, i.e., $t \notin Q$.

1. Case: $t = (2, 2, 4)$ is caught by the third predicate; leads to new cursor with $ql = (2, 3, 2)$

2. Case: $t = (2, 3, 1)$ is caught by the second predicate; leads to new cursor with $ql = (2, 3, 2)$

3. Case: $t = (2, 3, 4)$ is caught by the second predicate; leads to new cursor with $ql = (3, 2, 2)$

4. Case: $t = (3, 1, 1)$ is caught by the first predicate; leads to new cursor with $ql = (3, 2, 2)$

◇

11.5 Analysis of Skipper

In this section, we first present a CPU cost and I/O cost analysis in comparison to the standard range query processing on composite keys. Then we discuss in what scenarios Skipper even reaches the performance of multidimensional access methods.

11.5.1 Skipper and standard composite key processing

11.5.1.1 I/O and CPU analysis

Given the description of the Skipper technique and its optimization in the section above, we conclude that Skipper requires in worst case the same number of data

page accesses as the standard algorithm. Skipper never accesses data pages that are not accessed by the standard algorithm. With respect to CPU performance, we have to analyze the CPU overhead of skipping. The worst case for Skipper occurs when it accesses the same number of pages as the standard algorithm and at the same time all pages are reached via a skip. More precisely, there is one skip per page leading to the first tuple of the following page. Thus, Skipper has to perform one nextJumpIn computation for each accessed page. Combined with comparison operation of the post-filtering, however, this computation does not cause any overhead. Consequently, the Skipper technique never performs worse than the standard composite key query algorithm for B-Trees. On the other hand, the benefits of Skipper are significant: each page access saved by Skipper not only improves I/O performance but also reduces the CPU cost by saving post-filtering for all tuples on the skipped page.

11.5.1.2 When skipping gains

Now that we know that Skipper never performs worse than the standard algorithm, the question is when does Skipper gain substantially? As with multidimensional problems in general, no general answer can be given to this question as the performance of Skipper is strongly influenced by the nature of the multi-attribute restriction and the data distribution. However, we are presenting some heuristics that indicate when Skipper performs especially good. As usually assumed in the literature, indexes on composite keys are useful only for answering queries that restrict a prefix of the composite key to single values. Assuming a composite key $A_1 \circ A_2$, queries that either restrict A_1 or $A_1 \circ A_2$ can be answered efficiently with this index, but not queries that only restrict A_2. However, this is only true, if many different values of A_1 are stored in the database. The picture changes, however, if the number of distinct values of the lead attribute A_1 is small. Let us use $|\Pi_A(R)|$ to denote the number of distinct values of attribute A in relation R. Analogously, $|\Pi_{A_1 \circ \ldots \circ A_k}(R)|$ denotes the number of distinct values of the attribute combination $A_1, \ldots, A_k$ in R, and $|R|$ the number of records in R. Let $A_1, \ldots, A_d$ be the composite key of relation R that is stored on P pages with a page capacity C. Skipper performs better than the standard algorithm in terms of I/O if there is a high probability that pages can be skipped. The probability is higher when the intervals specified by prefixes of the composite key are larger than a disk page. Therefore, the Skipper technique has a positive effect for all key prefixes $A_1, \ldots, A_k$ that satisfy the condition:

$$\frac{|R|}{|\Pi_{A_1 \circ \ldots \circ A_k}(R)|} \geq C$$

For other prefixes, the benefit of Skipper depends on the data distribution and individual queries, but in any case Skipper will always be as good as the standard algorithm.

11.5.2 Skipper and multidimensional indexing

As we have mentioned earlier, Skipper corresponds to the UB-Tree range query algorithm specialized for the composite key orders. So from the algorithmic view there is no difference between a UB-Tree and a standard B-Tree using the Skipper technique. The data structures differ only in the clustering of the tuples. Thus, the performance strongly depends on the queries as we will examine in the next section.

11.6 Performance comparison

To validate our theoretical evaluation we conduct various performance comparisons with the GfK11D data set. Figure 11.6 shows the DDL statement for the used table.

```
Create table "GfK11D"(
   YEAR_ID integer,
   MONTH4_PERIOD_ID integer,
   MONTH2_PERIOD_ID integer,
   COUNTRY_ID integer,
   REGION_ID integer,
   MICROMARKET_ID integer,
   OUTLET_KEY integer,
   SECTOR_ID integer,
   CATEGORY_ID integer,
   PRODUCTGROUP_ID integer,
   ITEM_ID numeric(30,0),
   PD_PRICE integer,
   PD_SALES integer,
   PD_TURNOVER integer
) key is SECTOR_ID, YEAR_ID, COUNTRY_ID, CATEGORY_ID,
MONTH4_PERIOD_ID, REGION_ID, PRODUCTGROUP_ID,
MONTH2_PERIOD_ID, MICROMARKET_ID, ITEM_ID, OUTLET_KEY;
```

Figure 11.9: DDL statement for GfK11D fact table

The created fact table with the 43 million records has the size of 3011283 pages of 2KB each, i.e., about 5.7 GB.

11.6.1 Comparing Skipper to standard composite key processing

We first show the performance of the Skipper algorithm in comparison to the standard B-Tree algorithm. Using real queries from GFKDW we evaluate the influence of the prefix length resulting from the query predicates on the performance of the two techniques. The following queries restrict the hierarchy levels of the

- Product dimension down to the product group level to a point; the ITEM_ID is not restricted

- Segment dimension down to the region level to a point; MICROMARKET_ID and OUTLET_KEY are not restricted
- Time dimension:
  - Down to the MONTH2_PERIOD level to a point (QS1), resulting in a prefix length of 8
  - Down to the MONTH4_PERIOD level to a point (QS2), resulting in a prefix length of 7
  - Down to the YEAR level to a point (QS3), resulting in a prefix length of 4
  - Not at all (QS4), resulting in a prefix length of 1

The query suites QS1 - QS4 contain all 461 possible combinations of the various restrictions on the key attributes.

QS1 (Figure 11.10) already demonstrates that for skipping the number of page I/Os, and correspondingly the response time, is proportional to the result set size. On average, the I/Os are reduced by a factor of 7 and a maximal reduction factor of 190 is achieved. In contrast to that, the standard algorithm is rather insensitive to result set size (it is actually sensitive to the leading attribute restriction instead).

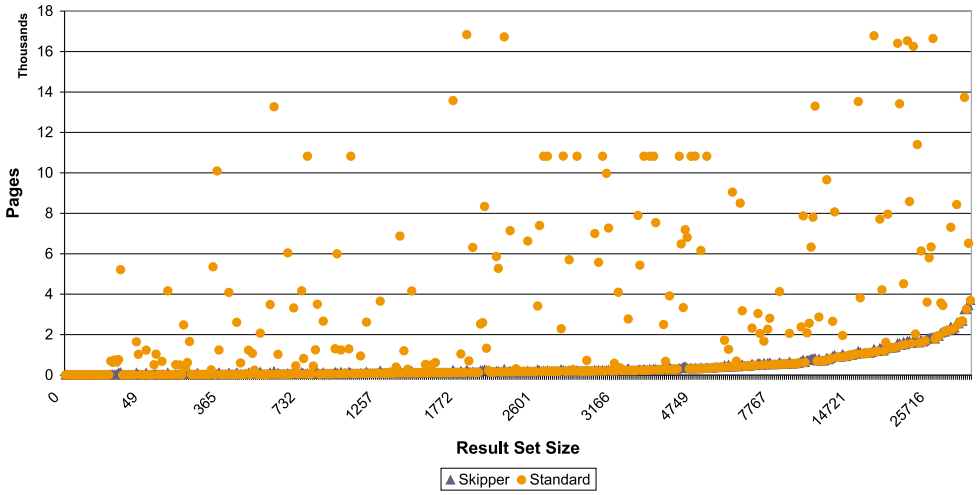


Figure 11.10: QS1: Restriction to Month2-Period level (prefix length = 8)

The behavior is similar for QS2, where the length of the restricted prefix is reduced from 8 (QS1) to 7 attributes (QS2). The number of skips does not increase, as still one interval (now covering two 2-months periods) has to be processed on the MONTH2-PERIOD level.

When reducing the restriction in the Time dimension further in QS3 (resulting in a prefix of length 4), Skipper can skip more and more intervals. For such queries,

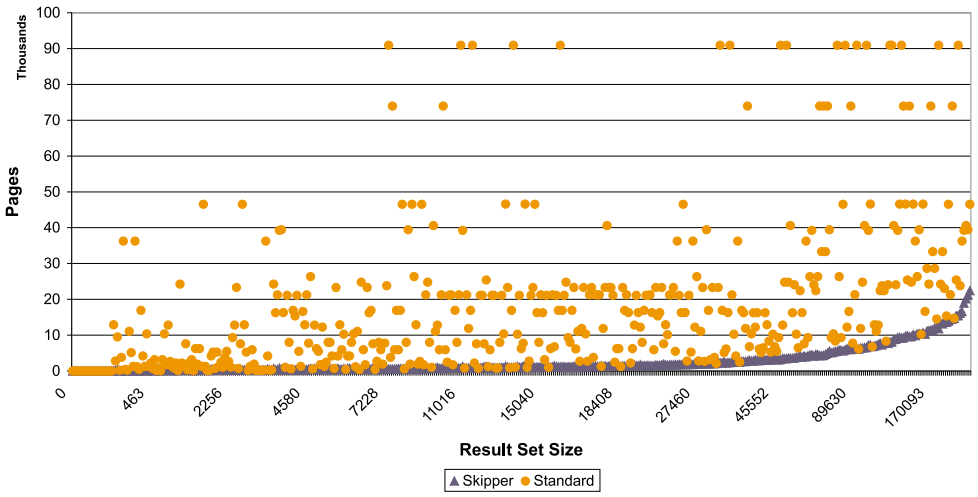


Figure 11.11: QS3: Restriction to Year level (prefix length = 4)

index skipping clearly outperforms the standard algorithm, requiring only $\frac{1}{19}$ of page accesses on average and down to $\frac{1}{465}$ I/O pages for some queries (see Figure 11.11).

This trend continues for queries with no restriction on the Time dimension at all (cf. Figure 11.12), i.e., reducing the length of the specified prefix to 1. On average, index skipping requires about $\frac{1}{600}$ page access (with a maximum speedup factor of more than 10.000) of the standard query processing. For QS4 the standard algorithm is already outperformed by a full table scan (FTS). A FTS has to read all of the 3.011.283 pages but can benefit from sequential access, reducing the number of random I/Os by the prefetching factor. At the same time, FTS significantly increases the CPU cost as each tuple has to be post-filtered.

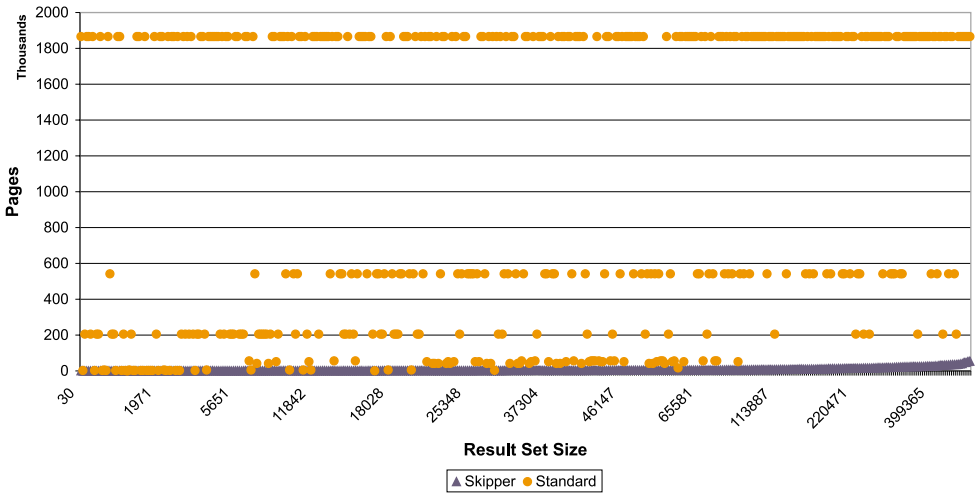


Figure 11.12: QS4: No restriction on Time dimension (prefix length=1)

Finally, we take a closer look to the influence of the number of skips on the speed-up of Skipper. Figure 11.13 shows the speed-up of Skipper compared to the standard algorithm for all queries: the horizontal axis is sorted by number of skips and then by the result set size; the y-axis has logarithmic scale. As shown theoretically, if no skips are performed, index skipping has the same I/O as the standard algorithm. As soon as skipping takes place, significant I/O savings are observed (up to a factor of 10.000!). This also shows that the CPU overhead of index skipping can be neglected

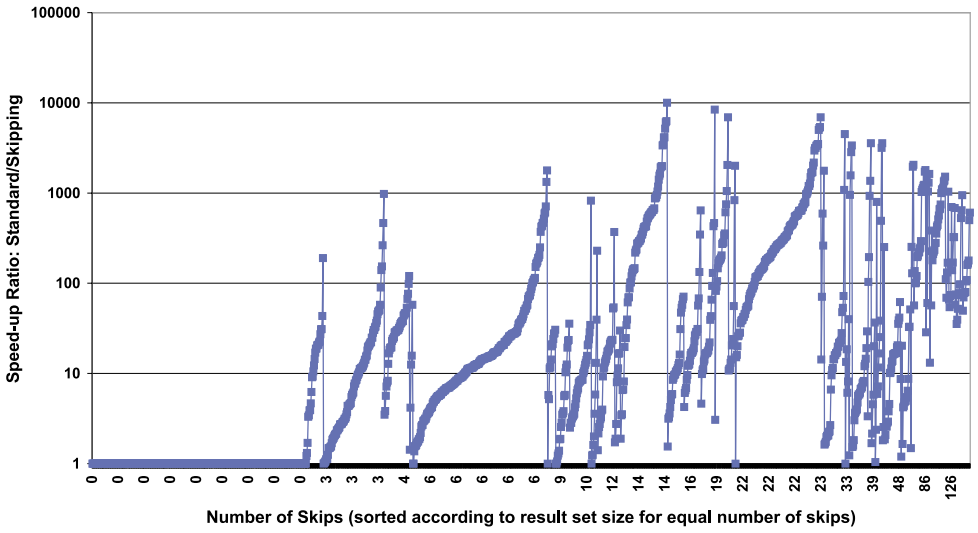


Figure 11.13: Relationship between number of skips and speed-up

for these practical scenarios: the significant savings (in the range of several thousand pages) in I/O cost were achieved by less than 150 skips, comparable in cost to only few page accesses.

11.6.2 Comparing Skipper to other access methods

After comparing Skipper to the standard composite key processing in the previous section we now analyze how Skipper performs compared to other access methods suitable for multi-attribute restrictions. To this end we compare Skipper to:

- standard algorithm
- multiple single-attribute indexes (MULT) in combination with index intersection: In our case we use secondary B-Trees; the tuple identifier lists of the single indexes are filtered by sorting and merging and thus it is guaranteed that each page is only accessed once.
- the UB-Tree (UB)
- the R*-Tree (RST)

For this measurement we use a subset of GfK11D containing 8,4 million records. All data structures have been bulk-loaded with appropriate methods resulting in comparable sizes. For the query workload, we adapted the query sets QS1, QS2, QS3, and QS4 for the smaller database size by ignoring queries that produce empty result sets. We analyze the behavior of the access methods depending on the length of the restricted key prefix. Figure 11.14 shows the results for the various index structures for QS1, i.e., queries with long restricted prefixes, as a box-plot.

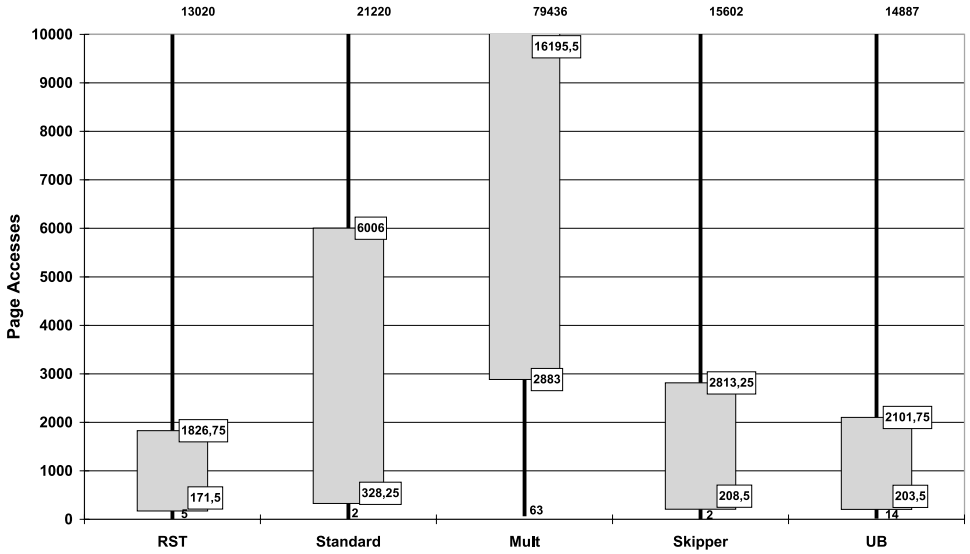


Figure 11.14: Comparison of various access methods for QS1 (long prefix restriction)

For QS1, the long prefix restriction leads to very selective queries, i.e., queries with small result sets. Skipper and the Standard algorithm benefit from the long key prefix restriction and both almost match the performance of the two multidimensional index structures. Only the non-clustering MULT scheme cannot compete with the other access methods. This access method has to retrieve much more pages, as tuples that are accessed together are not stored together.

The picture changes if we have no restriction on the prefix, i.e., if there is no restriction on the first attribute of the composite key. In this situation, Standard deteriorates into a full table scan leading to the worst performance. The performance of Skipper depends on the number and the position of the other key restrictions. If the non-restricted prefix is very long, Skipper will also turn into a full scan of the index and thus behave like the standard algorithm (first query in Figure 11.15). If the non-restricted prefix is short, Skipper performs better than the standard technique and, depending on the restriction, even reaches the performance of multidimensional indexes (second query in Figure 11.15) or will be outperformed by these (third query in Figure 11.15).

Finally, for short prefix restrictions (e.g., just the first two attributes of the composite key), and no further restrictions to key attributes, Skipper and Standard even outperform the multidimensional access methods (cf. Figure 11.16). In such

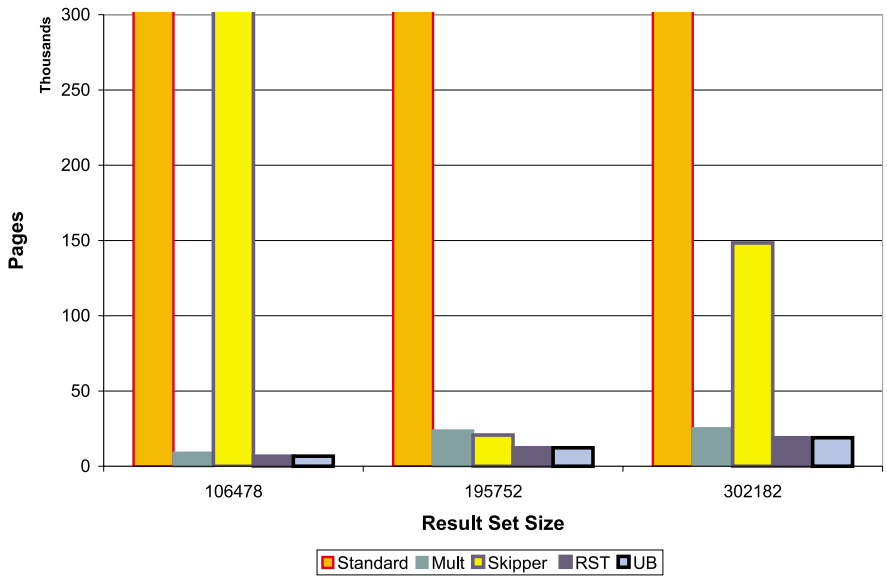


Figure 11.15: Non-prefix restriction

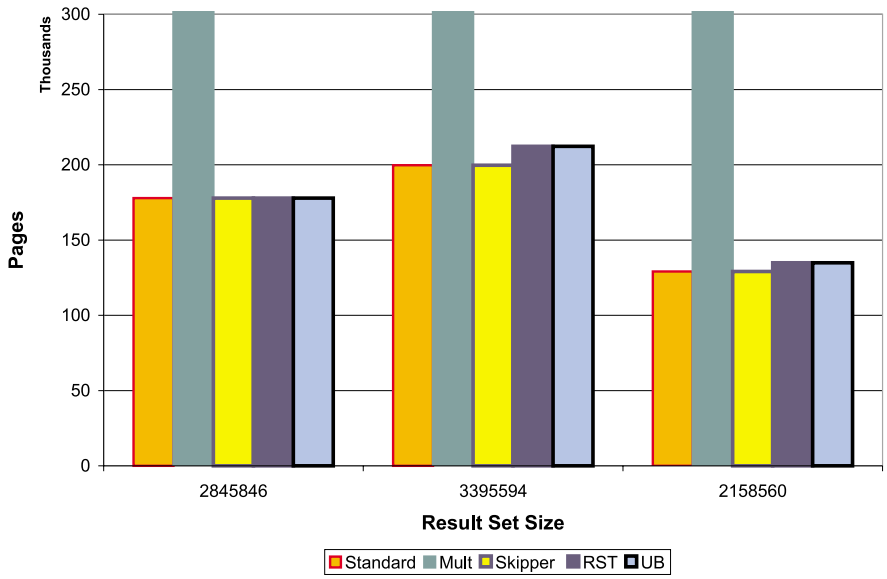


Figure 11.16: Short prefix restriction

cases, the clustering of the composite index is optimal, as a complete range of pages can be read sequentially.

11.7 Chapter notes and related work

In this chapter we have presented Skipper, an efficient technique to process multi-attribute restrictions on composite keys. Composite key indexes are heavily used in database applications as they are often the only way to get clustered access to data. However, the standard query processing algorithms only utilize restrictions on a fully specified prefix of the key, leading to poor performance when other query patterns arise. Skipper overcomes those deficiencies, making composite key indexes much more valuable. In this work, we have described the Skipper technique in detail and have compared it to the standard algorithm, both, analytically and experimentally. We have shown that Skipper is a true winner over the standard algorithm: Skipper never performs worse than the standard algorithm and potentially outperforms it by multiple orders of magnitude. The benefit of Skipper depends largely on the data distribution and the query profiles. Our measurements on a real world data warehouse have shown average speed-ups in the range of 100 - 600 and maximum speed-ups of up to 10.000. The resulting impact on query processing is significant: for many queries for which optimizers favor a costly full-table scan over a composite key index access by now, Skipper will provide a far more efficient solution, leading to efficient query processing and less resource consumption. Comparisons with multiple single-attribute indexes and multidimensional indexes have shown that Skipper is a strong competitor in this field. For favorable key orders, i.e., good clustering, Skipper can match the performance of multidimensional indexes. Still, it lacks the flexibility and robustness of those access methods as it depends on the chosen key order. The most important advantage of Skipper is that it is a true optimization of the standard algorithms that almost comes for free: Skipper requires only minor modifications of existing implementations. Thus, existing DBMSs can be easily extended to handle multi-attribute restrictions more efficiently.

Related work

B-Trees [BM72, Com79] are the de facto indexing structure for large data sets. Using composite keys, or even multiple indexes on one table to speed up query processing has been discussed in many places [GHRU97, OQ97, Sar97]. Range query processing on composite B-Trees is traditionally handled by the standard algorithm [GR93] as described in Section 11.1. The problem resulting from the necessity of a leading prefix restriction on the composite key is identified in various publications, e.g., [GHRU97, Ram98, SKS97]. However, to the best of our knowledge, [LJBY95] is the first and only work to introduce an advanced processing technique for multi-attribute restrictions on a composite key B-tree. The basic idea of Skipper is sketched but detailed algorithmic descriptions as well as a detailed analysis are missing.

Chapter 12

Summary

In our research work presented in this thesis, we address the complex problem of multidimensional indexing. Driven by the observation that despite a large body of research work the support in commercial systems is still weak, we raise the question of what is necessary to overcome this deficit. In the search for a solution, we have assessed if the *universal* B-Tree (UB-Tree) lives up to its name and may become a standard for multidimensional indexing, like the B-Tree for one-dimensional cases. Before we come to our conclusions, we briefly summarize our contributions.

As starting point, we define a comparison framework for access methods. In addition to the important aspect of query performance, we include properties that are important in practical applications as well, comprising for example space complexity, multi-user support, and flexibility. Based on this framework we carry out a comprehensive comparison of R*-Trees and UB-Trees.

In the second part of this thesis, we deal with the integration of the UB-Tree into a DBMS kernel. We present the basic algorithms, particularly an optimization of the range query algorithm with the crucial computation of the next-jump-in and next-jump-out points. Further, we cover the essential integration with the query optimizer, focussing on the central problem of cardinality estimation. We propose a new multidimensional histogram, the $\mathcal{Z} - Histogram$, based on the concept of UB-Trees and point out the limitation of histogram-based estimation techniques.

Finally, we generalize the address calculation scheme of UB-Trees to allow for arbitrary weighting of dimensions. We analyze how the address calculation scheme influences the clustering of the data, and so specifies the preferences among dimensions. We elaborate further on the special case of composite keys, leading to an advanced range query processing algorithm for standard B-Trees.

All of our findings are backed by theoretical analysis and experimental evaluation on real-world data sets.

12.1 Conclusion

For the evaluation of our findings, we recall the requirements for a general-purpose, multidimensional index stated in the introduction: universality, multi-user support,

symmetry w.r. to all dimensions, easy integration, dynamic behavior, worst case guarantees, and low storage complexity.

The results of the comparison of UB-Trees and R*-Trees clearly indicate that the UB-Tree outperforms the R*-Tree with respect to most criteria. It requires less space and provides better performance for the majority of queries. The most important disadvantage of R*-Trees is their inefficient behavior in dynamic environments, making it unsuitable for a wide range of applications. One advantage of the R*-Tree is the flexibility with respect to different data types: the concept of MBBs allows for handling of extended objects. For the UB-Tree various methods for supporting extended objects (e.g., for example the dual-space approach or approximation with intervals) are currently investigated, with promising initial results [Fen02].

Our experiences with the kernel integration of UB-Trees demonstrate another essential advantage: relying on the standard B-Tree as basic data structure reduces the complexity of the required extensions significantly. One can revert to proven solutions for system critical issues like locking and recovery, and can concentrate on the key algorithms. The coupling of the new access method with the query optimizer, i.e., the generation of plans recognizing the new index and operators, is facilitated by the modular design of modern optimizers. The problem to be solved is the estimation of the costs of the new methods, i.e., the cardinality estimation of the output of a query. As the cardinality depends on the data distribution no general model can be provided: one will always find pathological cases in which the model will not hold, i.e., produce estimation errors. Consequently, one has to rely on approximations based on data synopses. Current techniques, like histograms or sampling, work well for one-dimensional data, but often suffer from the *curse of sparse universes* in case of multidimensional data. As it seems impossible to find a general solution to the problem of getting a good cardinality estimation within a tolerable time for arbitrary queries, recent approaches try to improve the estimation for the current query focus. For most applications this will work well in practice.

Finally, the extension of the address calculation concept to weighted dimensions allows for broader usability of the index. There is now the possibility to fine tune the UB-Tree exactly for a given workload profile, making the construction of extra indexes dispensable in many cases.

Putting all results together, we conclude that the UB-Tree really deserves its name: it is closer to the notion of a universal, multidimensional index than any proposed method so far. The UB-Tree combines efficient query processing on multidimensional data with low space complexity and high dynamism. The relatively low effort of integration compared to other approaches makes it the prime candidate for a standard multidimensional index. The support for point data as well as for extended objects and the possibility to specify preferences among the indexed dimensions make the UB-Tree suitable for a wide range of applications.

12.2 Future work

Concluding our work, we want to point out future research directions, as far as we have not done it in the chapter notes, yet.

12.2.1 Approximate query processing with UB-Trees

With the ever increasing data volumes, query processing times are also increasing. Especially for interactive applications this leads to intolerable response times. On the other side, in many applications one does not really need the exact result at first, but can work with an approximate answer. For example, in the GfK data warehouse it sometimes is not necessary to know the total number of sales, say 1.043.000, but it suffices to know that the sales are above 1.000.000. Query processing techniques for such approximate answers have recently gained increasing attention. Evaluating such techniques for UB-Trees is therefore an interesting problem, and the idea of approximating the Z-regions with MBBs may be a good starting point for such research.

12.2.2 Advanced query types

Much room for future research is in the area of advanced query algorithms. New applications cause new query patterns that can not always be mapped to multidimensional range queries. Finding special algorithms for such problems for the UB-Tree has high potential of significantly increasing the efficiency of the index structure. One example are so-called *Skyline* queries [BKS01] that have gotten more attention just recently. In [KRR02], we propose a general solution for online Skyline queries that is based on nearest neighbor search, such that it can be implemented on-top of any multidimensional index structure supporting nearest neighbor queries. It will be interesting to see, if the Z-order of the UB-Tree can be exploited to find a even more efficient solution to Skyline queries.

12.2.3 Variable encoding of dimensions

Until now we have considered fixed sized domains for the dimensions. While this is a viable assumption, it may lead to problems in practical cases. As we have argued in Chapter 9 in the context of histograms, there is often a large difference between the size of the nominal and actual domain in real-world applications. The GfK DW provides a good example: the Product key is represented by an integer but only about 500.000 products are recorded. The length of the bit-representation required for UB-Tree indexing, however, depends on the nominal domain making it often much larger than actually required. On the other side, if one chooses a too small nominal domain one runs into the problem of reorganization in the case the domain overflows, i.e., if one wants to store more dimension members than the domain size (*enlarging the domain*). This problem is especially important in the context of hierarchy encoding as a fixed fan-out per hierarchy level has to be chosen.

Consequently, updating the hierarchy (known as *(slowly) changing dimensions* in the OLAP field) may cause expensive reorganization if a level overflows.

The open question is if one can find a flexible encoding for a domain that adapts automatically to increasing domain size. One idea is to explicitly encode the length of a dimension in the address, but it still has to be examined how this affects the CPU cost of the operations.

Appendix A

Data distribution of GfK3D

Having the GfK DW schema in mind, we take a look at how the data provided by GfK is distributed within the data cube. This will help to understand the various performance measurements on this warehouse. The results of a thorough analysis are presented in the form of diagrams where a vertical bar indicates the portion of facts associated with a particular hierarchy element (e.g., a certain year). Elements are suppressed if no fact records at all are associated with them.

A.1 Time dimension

Figure A.1, Figure A.2, and Figure A.3 show the data distribution in the Time dimension. The distribution over two-month periods is as close as the data ever comes to a uniform distribution for any attribute. The fact that the first two-month period of Year 1996 and fifth and sixth two-month periods of Year 1998 (see Figure A.3) are missing has an obvious effect on the higher levels in this dimension.

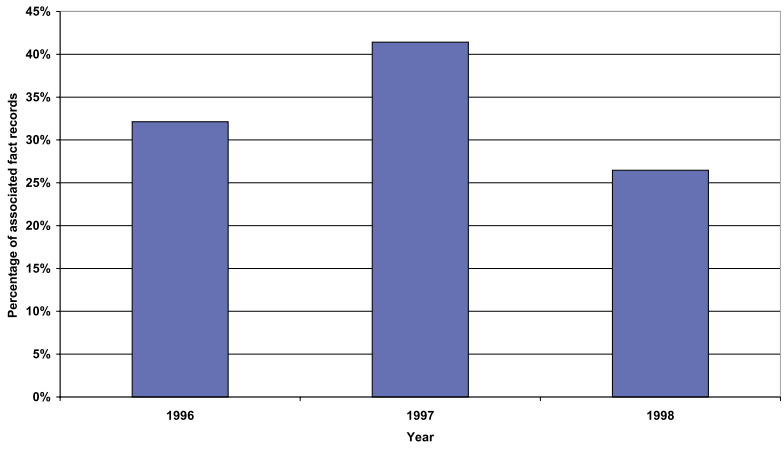


Figure A.1: Data distribution on Year level

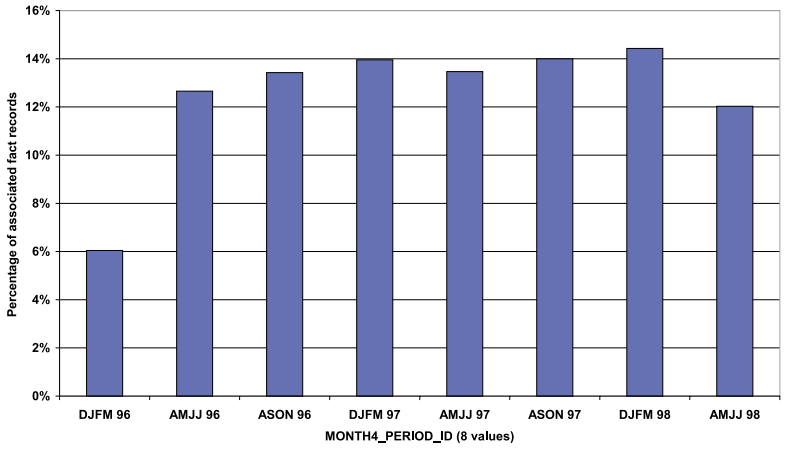


Figure A.2: Data distribution on 4-Month-Period level

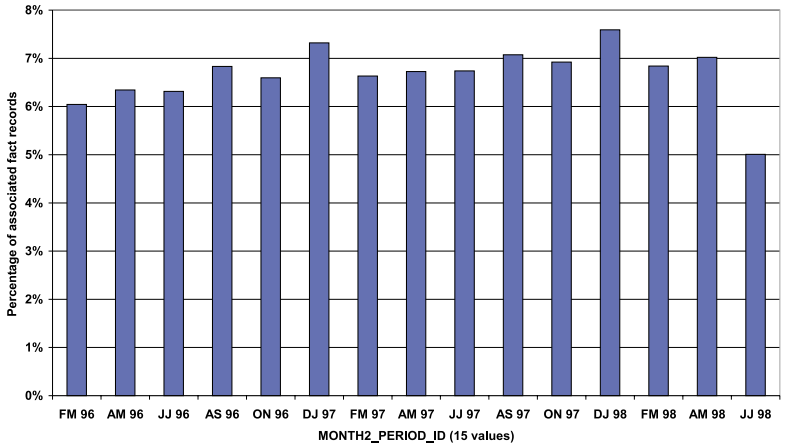


Figure A.3: Data distribution on 2-Month-Period level

A.2 Product dimension

For the Product Dimension, the portion of fact records belonging to a specific product group is depicted in Figure A.4. The individual values are of less interest than the fact that, although the most predominant product groups account for up to 812 950 or 1,90% of the records, which is more than ten times the mean percentage (0,16%), a restriction to a single product group will never yield more than roughly two percent of the data volume. A worst-case selectivity of less than two percent promises considerable effects of suitable indexing. Worst-case selectivity on the next higher level of this hierarchy path, product categories, is a lot higher, yet still within bounds for performance gains via indexing. Figure A.5 shows that portions can be as high as approximately 18%, where the average is 3,57%. Grouping the first seven categories as a single sector and assigning each of the last nine to a separate one are the main reasons for the distribution over product sectors, as depicted in Figure A.6.

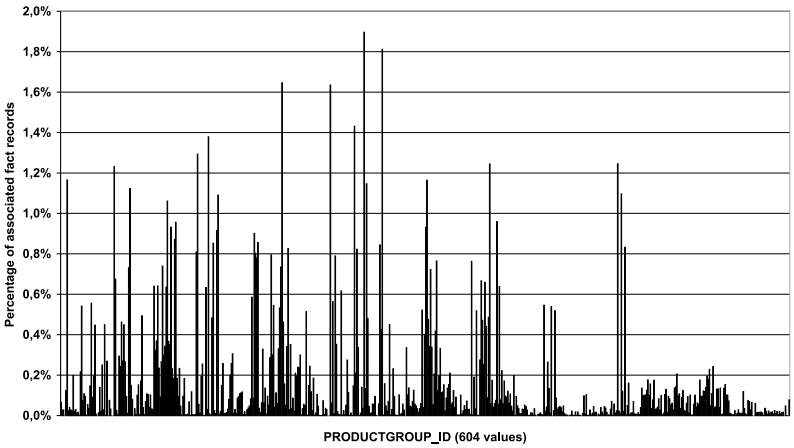


Figure A.4: Data distribution on Product Group level

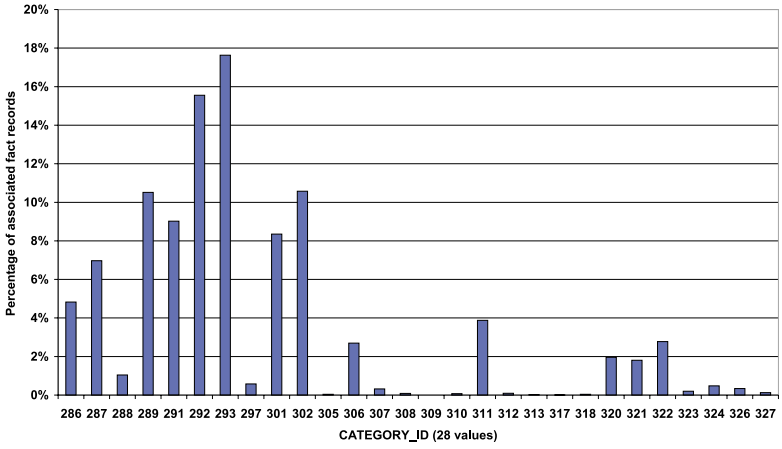


Figure A.5: Data distribution on Category level

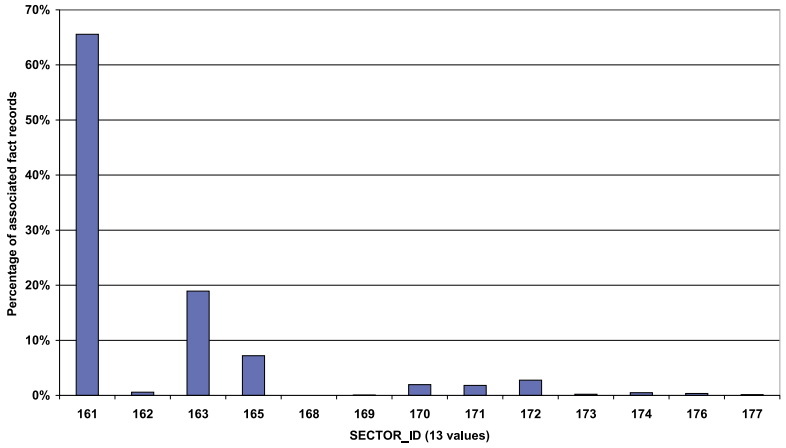


Figure A.6: Data distribution on Sector level

A.3 Segment dimension

Finally, we consider the Segment dimension. Figure A.7 illustrates the fact distribution over countries. It just happens that GfK collects census data in Country 18, thus this country dominates with respect to the number of fact records collected. The same holds for the region level shown in Figure A.8: Country 18 is only assigned one region, leading to a selectivity of more than 30% for a restriction on the region level in worst case.

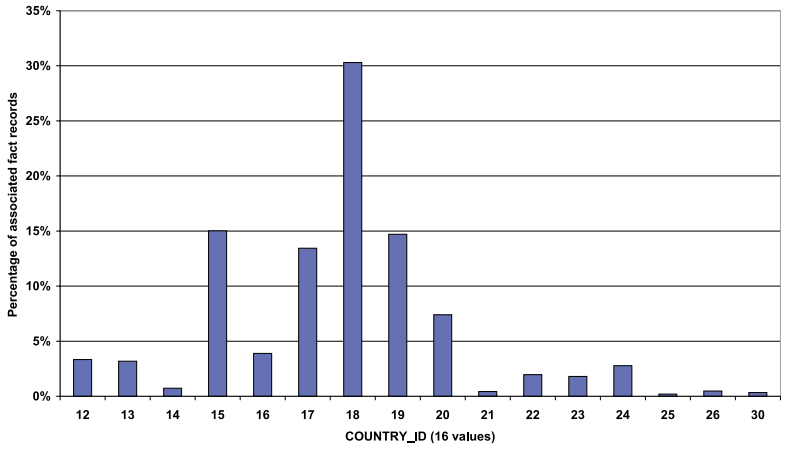


Figure A.7: Data distribution on Country level

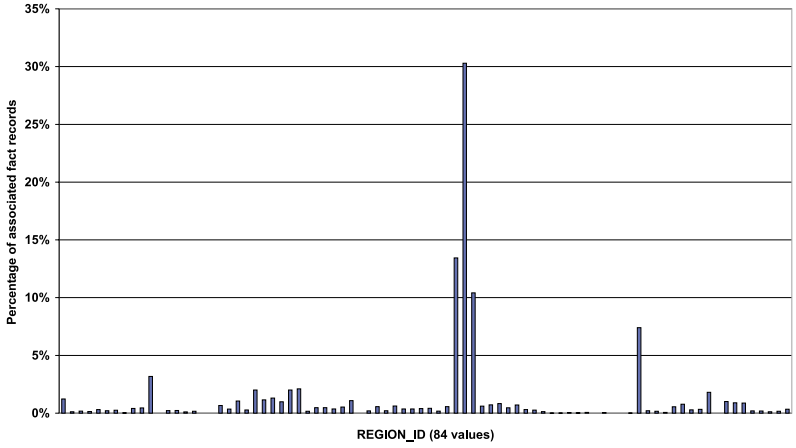


Figure A.8: Data distribution on Region level

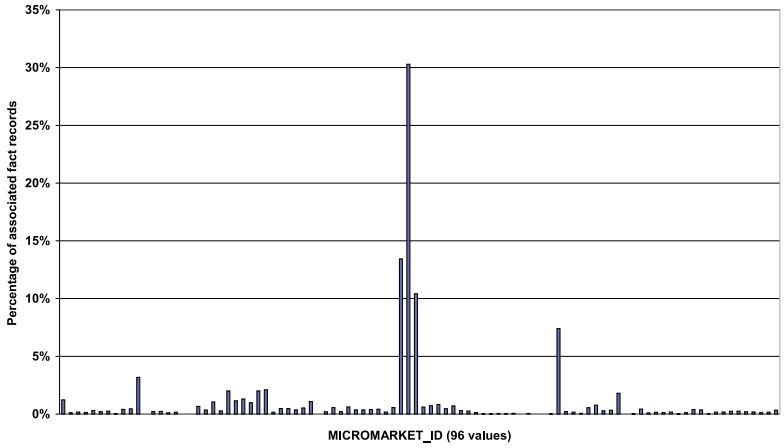


Figure A.9: Data distribution on Micromarket level

Bibliography

[AHVV99] Lars Arge, Klaus Hinrichs, Jan Vahrenhold, and Jeffrey Scott Vitter. Efficient Bulk Operations on Dynamic R-Trees. In *Proc. of ALENEX*, 1999.

[Ant92] Gennady Antoshenkov. Random Sampling from Pseudo-Ranked B+-Trees. In Li-Yan Yuan, editor, *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings*, pages 375–382. Morgan Kaufmann, 1992.

[Aok98a] Paul M. Aoki. Algorithms for Index-Assisted Selectivity Estimation. Technical Report UCB//CSD-98-1021, Computer Science Division (EECS), University of California, Berkeley, California 94720, October 1998.

[Aok98b] Paul M. Aoki. Generalizing "Search" in Generalized Search Trees (Extended Abstract). In *Proceedings of the Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA*, pages 380–389. IEEE Computer Society, 1998.

[Aok99] Paul M. Aoki. Algorithms for index-assisted selectivity estimation. In *Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Austrialia*, page 258. IEEE Computer Society, 1999.

[AZ96] Gennady Antoshenkov and Mohamed Ziauddin. Query processing and optimization in oracle rdb. *VLDB Journal*, 5(4):229–237, 1996.

[Bay96] Rudolf Bayer. The universal B-Tree for multidimensional Indexing. Technical Report TUM-I9637, Institut für Informatik, TU München, 1996.

[Bay97] Rudolf Bayer. The universal B-Tree for multidimensional Indexing: General Concepts. In *World-Wide Computing and its Applications '97 (WWCA '97), Lecture Notes on Computer Science*. Springer Verlag, 1997. Tsukuba, Japan.

[BBK98] Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. In

Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 142–153. ACM Press, 1998.

[BCG01] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. STHoles: A Multidimensional Workload-Aware Histogram. In *SIGMOD 2001, Proceedings ACM SIGMOD International Conference on Management of Data, Santa Barbara, California, USA*, 2001.

[BKS01] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The Skyline Operator. In *Proc. of ICDE, Heidelberg, Germany*, 2001.

[BKSS90] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient robust access method for points and rectangles. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 19(2):322–331, June 1990.

[BM72] Rudolf Bayer and E. McCreight. Organization and Maintainance of large ordered Indexes. In *Acta Informatica 1*, pages 173–189, 1972.

[BMK99] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 54–65. Morgan Kaufmann, 1999.

[BMR01] Philip Bohannon, Peter Mcllroy, and Rajeev Rastogi. Main-Memory Index Structures with Fixed-Size Partial Keys, 2001.

[BSSJ99] Rasa Bliujute, Simonas Saltenis, Giedrius Slivinskas, and Christian S. Jensen. Developing a DataBlade for a New Index. In *Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Austrialia*, pages 314–323. IEEE Computer Society, 1999.

[BU77] Rudolf Bayer and Karl Unterauer. Prefix B-Trees. *TODS*, 2(1):11–26, 1977.

[CCF+99] Weidong Chen, Jyh-Herng Chow, You-Chin Fuh, Jean Grandbois, Michelle Jou, Nelson Mendonça Mattos, Brian T. Tran, and Yun Wang. High level indexing of user-defined types. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 554–564. Morgan Kaufmann, 1999.

[CD97] Surajit Chaudhuri and Umeshwar Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, 26(1):65–74, 1997.

[CGRS00] Kaushik Chakrabarti, Minos N. Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Approximate Query Processing Using Wavelets. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 111–122. Morgan Kaufmann, 2000.

[Cha98] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington*, pages 34–43. ACM Press, 1998.

[CI98] Chee Yong Chan and Yannis E. Ioannidis. Bitmap Index Design and Evaluation. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 355–366. ACM Press, 1998.

[CM99a] Kaushik Chakrabarti and Sharad Mehrotra. The hybrid tree: An index structure for high dimensional feature spaces. In *Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Austrialia*, pages 440–447. IEEE Computer Society, 1999.

[CM99b] Surajit Chaudhuri and Rajeev Motwani. On sampling and relational operators. *IEEE Data Engineering Bulletin*, 22(4):41–46, 1999.

[CM00] Kaushik Chakrabarti and Sharad Mehrotra. Local dimensionality reduction: A new approach to indexing high dimensional spaces. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 89–100. Morgan Kaufmann, 2000.

[CN97] Surajit Chaudhuri and Vivek R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 146–155. Morgan Kaufmann, 1997.

[CN98a] Surajit Chaudhuri and Vivek R. Narasayya. Autoadmin 'what-if' index analysis utility. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 367–378. ACM Press, 1998.

[CN98b] Surajit Chaudhuri and Vivek R. Narasayya. Microsoft index tuning wizard for sql server 7.0. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 553–554. ACM Press, 1998.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6):377–387, 1970.

[Col96] George Colliat. Olap, relational, and multidimensional database systems. *SIGMOD Record*, 25(3):64–69, 1996.

[Com79] Douglas Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[dBSW97] Jochen Van den Bercken, Bernhard Seeger, and Peter Widmayer. A Generic Approach to Bulk Loading Multidimensional Index Structures. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 406–415. Morgan Kaufmann, 1997.

[DGR01] Amol Deshpande, Minos N. Garofalakis, and Rajeev Rastogi. Independence is Good: Dependency-Based Histogram Synopses for High-Dimensional Data. In *SIGMOD 2001, Proceedings ACM SIGMOD International Conference on Management of Data, Santa Barbara, California, USA*, 2001.

[EDI] *http://edith.in.tum.de*. The EDITH Project.

[Fal86] Christos Faloutsos. Multiattribute Hashing Using Gray Codes. In Carlo Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 28-30, 1986*, pages 227–238. ACM Press, 1986.

[Fal88] Christos Faloutsos. Gray Codes for Partial Match and Range Queries. *IEEE TSE*, 14(10):1381–1393, 1988.

[Fen02] Robert Fenk. *Handling extended Objects with UB-Trees (Draft version)*. PhD thesis, Fakultät für Informatik an der Technischen Universität München, 2002.

[FK94] Christos Faloutsos and Ibrahim Kamel. Beyond Uniformity and Independence: Analysis of R-trees Using the Concept of Fractal Dimension. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 24-26, 1994, Minneapolis, Minnesota*, pages 4–13. ACM Press, 1994.

[FKM+00] Robert Fenk, Akihiko Kawakami, Volker Markl, Rudolf Bayer, and Shuichi Osaki. Bulk loading a Data Warehouse built upon a UB-Tree. In *Proc. of IDEAS Conf. 2000, Yokohama, Japan*, 2000.

[FMB99] Robert Fenk, Volker Markl, and Rudolf Bayer. Improving Multidimensional Range Queries of non rectangular Volumes specified by a Query Box Set. In *Proceedings of International Symposium on Database, Web and Cooperative Systems (DWACOS), Baden-Baden, Germany*, 1999.

[FMB00] Robert Fenk, Volker Markl, and Rudolf Bayer. Management and Query Processing of one dimensional Intervals with the UB-Tree. In *Ph.D. Workshop of EDBT 2000, Konstanz, Germany*, 2000.

[FNPS79] Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible Hashing - A Fast Access Method for Dynamic Files. *TODS*, 4(3):315–344, 1979.

[Fre87] Michael Freeston. The BANG File: A New Kind of Grid File. In Umeshwar Dayal and Irving L. Traiger, editors, *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987*, pages 260–269. ACM Press, 1987.

[FSR87] Christos Faloutsos, Timos K. Sellis, and Nick Roussopoulos. Analysis of Object Oriented Spatial Access Methods. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987*, pages 426–439. ACM Press, 1987.

[Gae95] Volker Gaede. Optimal Redundancy in Spatial Database Systems. In Max J. Egenhofer and John R. Herring, editors, *Advances in Spatial Databases, 4th International Symposium, SSD'95, Portland, Maine, USA, August 6-9, 1995, Proceedings*, volume 951 of *Lecture Notes in Computer Science*, pages 96–116. Springer, 1995.

[GG98] Volker Gaede and Oliver Günther. Multidimensional Access Methods. In *Computing Surveys 30(2)*, pages 170–231. ACM Press, 1998.

[GG01] Minos N. Garofalakis and Phillip B. Gibbon. Approximate query processing: Taming the terabytes. Tutorial of VLDB 2001, Roma, Italy, 2001.

[GHRU97] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Index selection for OLAP. In *Proceedings of ICDE 1997*, pages 208–219, 1997.

[GKTD00] Dimitrios Gunopulos, George Kollios, Vassilis J. Tsotras, and Carlotta Domeniconi. Approximating Multi-Dimensional Aggregate Range

Queries over Real Attributes. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*. ACM, 2000.

[GL01] Goetz Graefe and Paul Larson. B-Tree Indexes and CPU Caches. In *Proceedings of the Seventeenth International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, pages 349–358. IEEE Computer Society, 2001.

[GR93] Jim Gray and Andreas Reuter. *Transactional Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[GRS98] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In *Proceedings of the Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA*, pages 370–379. IEEE Computer Society, 1998.

[Gün89] Oliver Günther. The Design of the Cell Tree: An Object-Oriented Index Structure for Geometric Databases. In *Proceedings of the Fifth International Conference on Data Engineering, February 6-10, 1989, Los Angeles, California, USA*, pages 598–605. IEEE Computer Society, 1989.

[Gut84] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In Beatrice Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57. ACM Press, 1984.

[Hin85] Klaus Hinrichs. Implementation of the Grid File: Design Concepts and Experience. *BIT*, 25(4):569–592, 1985.

[HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized Search Trees for Database Systems. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 562–573. Morgan Kaufmann, 1995.

[HSW88] Andreas Hutflesz, Hans-Werner Six, and Peter Widmayer. Twin Grid Files: Space Optimizing Access Schemes. In Haran Boral and Per-Åke Larson, editors, *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, June 1-3, 1988*, pages 183–190. ACM Press, 1988.

[HSW89] Andreas Henrich, Hans-Werner Six, and Peter Widmayer. The LSD tree: Spatial Access to Multidimensional Point and Nonpoint Objects. In Peter M. G. Apers and Gio Wiederhold, editors, *Proceedings of the*

Fifteenth International Conference on Very Large Data Bases, August 22-25, 1989, Amsterdam, The Netherlands, pages 45–53. Morgan Kaufmann, 1989.

[IBM01] IBM. IBM Ultrastar 36Z15 hard disk drive data sheet, 2001.

[Inf99] Informix. *Informix Dynamic Server with Universal Data Option Version 9.1.X Documentation*. Informix Software Incorporation, 1999.

[IP95] Yannis Ioannidis and V. Poosala. Balancing Histogram Optimality and Practicality for Query Result Size Estimation. In *Proceedings of ACM Sigmod*, 1995.

[Jag90] H. V. Jagadish. Linear Clustering of Objects with Multiple Atributes. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 332–342. ACM Press, 1990.

[Jür99] Marcus Jürgens. *Index Structures for Data Warehouses*. PhD thesis, Fachbereich für Mathematik und Informatik an der Freien Universität Berlin, 1999.

[KB95] Marcel Kornacker and Douglas Banks. High-concurrency Locking in R-Trees. In *Proc. of VLDB, Zürich, Switzerland*, 1995.

[KCK01] Kihong Kim, Sang K. Cha, and Keunjoo Kwon. Optimizing Multidimensional Index Trees for Main Memory Access. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, May 21-24, 2001, Santa Barbara, California, USA*, 2001.

[KF93] Ibrahim Kamel and Christos Faloutsos. On Packing R-trees. In *CIKM*, pages 490–499, 1993.

[KMH97] Marcel Kornacker, C. Mohan, and Joseph M. Hellerstein. Concurrency and Recovery in Generalize Search Trees. In *Proc. of SIGMOD*, 1997.

[Knu98] Donald E. Knuth. *Sorting and Searching*. Addison - Wesley, 2 edition, 1998.

[Kor99] Marcel Kornacker. High-Performance Extensible Indexing. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 699–708. Morgan Kaufmann, 1999.

[KRR02] Donald Kossmann, Frank Ramsak, and Steffen Rost. Picking winners in the Sky: An Online Algorithm for Skyline Queries. In *submitted for publication*, 2002.

[KSSS90] H.-P. Kriegel, M. Schiwietz, R. Schneider, and B. Seeger. Performance Comparison of Point and Spatial Access Methods. In *Proc. 1st Symp. on the Design and Implementation of Large Spatial Databases, Santa Barbara, CA, 1989*, volume 409 of *Lecture Notes in Computer Science*, pages 89–114. Springer, 1990.

[LC86a] Tobin J. Lehman and Michael J. Carey. Query Processing in Main Memory Database Management Systems. In Carlo Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 28-30, 1986*, pages 239–250. ACM Press, 1986.

[LC86b] Tobin J. Lehman and Michael J. Carey. A study of index structures for main memory database management systems. In Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 294–303. Morgan Kaufmann, 1986.

[LEL97] Scott T. Leutenegger, J. M. Edgington, and Mario A. Lopez. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K*, pages 497–506. IEEE Computer Society, 1997.

[Lit80] Witold Litwin. Linear Hashing: A New Tool for File and Table Addressing. In *Sixth International Conference on Very Large Data Bases, October 1-3, 1980, Montreal, Quebec, Canada, Proceedings*, pages 212–223. IEEE Computer Society Press, 1980.

[LJBY95] Harry Leslie, Rohit Jain, Dave Birdsall, and Hedieh Yaghmai. Efficient Search of Multi-Dimensional B-Trees. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 710–719. Morgan Kaufmann, 1995.

[LS90] David B. Lomet and Betty Salzberg. The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance. *TODS*, 15(4):625–658, 1990.

[Mar99] Volker Markl. *Processing Relational Queries using a Multidimensional Access Technique*. PhD thesis, DISDBIS, Band 59, Infix Verlag, 1999.

[MD88] M. Muralikrishna and David J. DeWitt. Equi-Depth Histograms For Estimating Selectivity Factors For Multi-Dimensional Queries. In Haran Boral and Per-Åke Larson, editors, *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, June 1-3, 1988*, pages 28–36. ACM Press, 1988.

[MIS] *http://mistral.in.tum.de*. The MISTRAL Project.

[MRB99] Volker Markl, Frank Ramsak, and Rudolf Bayer. Improving OLAP Performance by Multidimensional Hierarchical Clustering. In *Proc. of IDEAS Conf., Montreal, Canada*, 1999.

[MS000] *SQL Server 2000 Books Online*. Microsoft Corporation, 2000.

[MZB99] Volker Markl, Martin Zirkel, and Rudolf Bayer. Processing Operations with Restrictions in RDBMS without External Sorting: The Tetris Algorithm. In *Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Austrialia*, pages 562–571. IEEE Computer Society, 1999.

[NHS84] Jürg Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *TODS*, 9(1):38–71, 1984.

[NK94] Vincent Ng and Tiko Kameda. The R-Link Tree: A Recoverable Index Structure for Spatial Data. In *Proc of DEXA 1994*, 1994.

[NNT00] Tapio Niemi, Jyrki Nummenmaa, and Peter Thanisch. Functional Dependencies in Controlling Sparsity of OLAP Cubes. In Yahiko Kambayashi, Mukesh K. Mohania, and A. Min Tjoa, editors, *Data Warehousing and Knowledge Discovery, Second International Conference, DaWaK 2000, London, UK, September 4-6, 2000, Proceedings*, volume 1874 of *Lecture Notes in Computer Science*. Springer, 2000.

[OG95] Patrick E. O'Neil and Goetz Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–11, 1995.

[OM84] Jack A. Orenstein and T. H. Merrett. A Class of Data Structures for Associative Searching. In *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, April 2-4, 1984, Waterloo, Ontario, Canada*, pages 181–190. ACM, 1984.

[OQ97] Patrick E. O'Neil and Dallan Quass. Improved query performance with variant indexes. In Joan Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 38–49. ACM Press, 1997.

[ORA99] *Oracle 8i Utilities / Documentation*. Oracle Corporation, 1999.

[Ore89a] Jack A. Orenstein. Redundancy in Spatial Databases. In James Clifford, Bruce G. Lindsay, and David Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989*, pages 294–305. ACM Press, 1989.

[Ore89b] Jack A. Orenstein. Strategies for Optimizing the Use of Redundancy in Spatial Databases. In Alejandro P. Buchmann, Oliver Günther, Terence R. Smith, and Y.-F. Wang, editors, *Design and Implementation of Large Spatial Databases, First Symposium SSD'89, Santa Barbara, California, July 17/18, 1989, Proceedings*, volume 409 of *Lecture Notes in Computer Science*, pages 115–134. Springer, 1989.

[Ore90] Jack A. Orenstein. A Comparison of Spatial Query Processing Techniques for Native and Parameter Spaces. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 343–352. ACM Press, 1990.

[PIHS96] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 294–305. ACM Press, 1996.

[Poo97] Viswanath Poosala. *Histogram-Based Estimation Techniques in Database Systems*. PhD thesis, University of Wisconsin, 1997.

[Ram98] Raghu Ramakrishnan. *Database Management Systems*. WCB/McGraw-Hill, 1998.

[Rit99] Roland Ritsch. *Optimization and Evaluation of Array Queries in Database Management Systems*. PhD thesis, Fakultät für Informatik der Technischen Universität München, 1999.

[RKV95] Nick Roussopoulos, Stephen Kelley, and Frédéic Vincent. Nearest neighbor queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 71–79. ACM Press, 1995.

[RMF+00] Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhardt, and Rudolf Bayer. Integrating the UB-Tree into a Database System Kernel. In *VLDB2000, Proceedings of International Conference on Very Large Data Bases, 2000, Cairo, Egypt*, 2000.

[RMF+01] Frank Ramsak, Volker Markl, Robert Fenk, Rudolf Bayer, and Thomas Ruf. Interactive ROLAP on Large Datasets: A Case Study with UB-Trees. In *IDEAS'01, Proceedings of the 2001 International Database Engineering & Applications Symposium, Grenoble, France, July 16-18, 2001*. IEEE, 2001.

[Rob81] John T. Robinson. The K-D-B-Tree: A Search Structure For large multidimensional dynamic Indexes. In Y. Edmund Lien, editor, *Proceedings*

of the 1981 ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan, April 29 - May 1, 1981, pages 10–18. ACM Press, 1981.

[RSV01] Philippe Rigaux, Michel Scholl, and Agnès Voisard. *Spatial Databases with Application to GIS*. Morgan Kaufmann Publishers, 2001.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In Philip A. Bernstein, editor, *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30 - June 1*, pages 23–34. ACM, 1979.

[Sag94] Hans Sagan. *Space-Filling Curves*. Springer Verlag, 1994.

[Sap01] Carsten Sapia. *PROMISE: Modeling and Predicting User Behavior for Online Analytical Processing Applications*. PhD thesis, Fachbereich für Informatik an der Technischen Universität München, 2001.

[Sar97] Sunita Sarawagi. Indexing olap data. *Data Engineering Bulletin*, 20(1):36–43, 1997.

[SK90] Bernhard Seeger and Hans-Peter Kriegel. The Buddy-Tree: An Efficient and Robust Access Method for Spatial Data Base Systems. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 590–601. Morgan Kaufmann, 1990.

[SKS97] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 3 edition, 1997.

[SLMK01] Michael Stillger, Guy Lohman, Volker Markl, and Mokhtar Kandil. LEO - DB2's Learning Optimizer. In Peter Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard Snodgrass, editors, *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB), Rome, Italy, September 11-14, 2001*, pages 19–28. Morgan Kaufmann, 2001.

[SRF87] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 507–518. Morgan Kaufmann, 1987.

[Str00] Michael Streichsbier. Design and Implementation of an algorithm linearizing multidimensional intervals and its application to selectivity estimation. Master's thesis, Technische Universität München, 2000.

[Sun96] Prakash Sundaresan. Data warehousing features in informix online xps (abstract). In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems, December 18-20, 1996, Miami Beach, Florida, USA*, page 288. IEEE Computer Society, 1996.

[SW88] Hans-Werner Six and Peter Widmayer. Spatial Searching in Geometric Databases. In *Proceedings of the Fourth International Conference on Data Engineering, February 1-5, 1988, Los Angeles, California, USA*, pages 496–503. IEEE Computer Society, 1988.

[Tam82] Markku Tamminen. The Extendible Cell Method for Closest Point Problems. *BIT*, 22(1):27–41, 1982.

[TAS01] *TransBase Documentation*. TransAction GmbH, 2001.

[TH81] H. Tropf and H. Herzog. Multidimensional Range Search in Dynamically Balanced Trees. *Angewandte Informatik*, 1981(2):71–77, 1981.

[Tic99] Anton Tichatschek. Implementation and Analysis of the Spiral Algorithm for Nearest-Neighbor Queries with UB-Trees. Master's thesis, Technische Universität München, 1999.

[TPC99] TPC. *TPC BENCHMARK H, Decision Support, Standard Specification, Revision 1.3.0*. Transaction Processing Performance Council (TPC), 1999.

[TS96] Yannis Theodoridis and Timos Sellis. A Model for the Prediction of R-Tree Performance. In *Proc. of PODS 1996, Montreal Quebec, Canada*, 1996.

[USC] *http://www.census.gov*. US Census Bureau.

[WB98] Ming-Chuan Wu and Alejandro P. Buchmann. Encoded Bitmap Indexing for Data Warehouses. In *Proceedings of the Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA*, pages 220–230. IEEE Computer Society, 1998.

[WSB98] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 194–205. Morgan Kaufmann, 1998.

[ZDNS98] Yihong Zhao, Prasad Deshpande, Jeffrey F. Naughton, and Amit Shukla. Simultaneous optimization and evaluation of multiple dimensional queries. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 271–282. ACM Press, 1998.

[Zir02] Martin Zirkel. *Bewertung des UB-Baums in der Anfragebearbeitung unter Berücksichtigung der Sortierung (Draft version).* PhD thesis, Fakultät für Informatik der Technischen Universität München, 2002.

[ZMB01] Martin Zirkel, Volker Markl, and Rudolf Bayer. Exploitation of Presortedness for Sorting in Query Processing: The TempTris-Algorithm for UB-Trees. In *IDEAS'01, Proceedings of the 2001 International Database Engineering & Applications Symposium, Grenoble, France, July 16-18, 2001*. IEEE, 2001.

[ZMR96] Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. Guidelines for presentation and comparison of indexing techniques. *SIGMOD Record*, 25(3):10–15, 1996.