Institut für Informatik

der Technischen Universität München

MISTRAL:
Processing Relational Queries using a Multidimensional Access Technique

Volker Markl

Preface

Classical one dimensional B-trees have been the standard access method of all commercial database systems for many years. This dissertation is a very promising effort to introduce universal B-trees (UB-trees), the multidimensional variant of B-trees, as a basic access method into the core of fundamental DB-technology. The universal relevance of UB-trees is a consequence of the fact, that every relation can be considered as a set of points in multidimensional space. UB-trees organize this space for efficient processing of the data, resulting for many types of queries in orders of magnitude improvement over classical methods.

The thesis lays the theoretical foundations of UB-tress, predicts their performance by analytical models and validates these models by experiments using large real world databases and real life applications and queries from the field of datawarehousing. The need to sort data and intermediate results frequently is an annoying drawback of today‘s query processing methods. In combination with the Tetris algorithm UB-trees allow to avoid sorting in most cases, leading to dramatic improvements of response time, storage requirement and overall query processing time.

Adding a new access method requires to consider all aspects of database systems:

- architecture of subsystems
- query optimization
- query processing
- multiuser operation and synchronization
- bulk loading
- storage requirement
- parallelism, etc.

Since UB-trees rely on classical B-trees for their implementation, all of these issues can be solved in a satisfactory way and can be dealt with elegantly.

The performance experiments reported in this thesis were carried out with an implementation of UB-trees as a middleware layer on top of SQL. Additional performance improvements can be gained by integrating the UB-tree technology in the kernel of database systems.

This thesis is a cornerstone of the MISTRAL project. MISTRAL has the goal to introduce UB-trees as a new access method into database systems with the fascinating vision, to extend fundamental database technology in an essential way. MISTRAL is financially supported by SAP, Teijin, NEC, Hitachi, the European Commission, Project MDA, TAS, Gfk and Microsoft.

Munich, July 25, 1999 Prof. Rudolf Bayer, Ph.D.

Institut für Informatik

der Technischen Universität München

MISTRAL: Processing Relational Queries using a Multidimensional Access Technique

Volker Markl

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. B. Brügge, Ph.D.
Prüfer der Dissertation:	1.	Univ.-Prof. R. Bayer, Ph.D.
	2.	Univ.-Prof. J.-C. Freytag, Ph.D., Humboldt-Universität zu Berlin

Die Dissertation wurde am 15.3.1999 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 29.6.1999 angenommen.

Title of the Thesis:

MISTRAL: Processing Relational Queries using a Multidimensional Access Technique

Author:

Dipl. Inform. Volker Markl

Supervisors:

1. Prof. Rudolf Bayer, Ph.D., Technische Universität München
2. Prof. Johann Christoph Freytag, Ph.D., Humboldt Universität Berlin

Keywords:

relational database management systems, query processing, query optimization
multidimensional access methods, indexing
data structures, B-Tree, UB-Tree
data warehousing, relational OLAP, OLTP
benchmarking, TPC-D

Abstract:

A multidimensional access method offering significant performance increases by intelligently partitioning the query space is applied to relational database management systems (RDBMS). We introduce a formal model for multidimensional partitioned relations and discuss several typical query patterns. The model identifies the significance of multidimensional range queries and sort operations. The discussion of current access methods gives rise to the need for a multidimensional partitioning of relations. A detailed analysis of space partitioning focussing especially on Z-ordering illustrates the principle benefits of multidimensional indexes. After describing the UB-Tree and its standard algorithms for insertion, deletion, point queries, and range queries, we introduce the spiral algorithm for nearest neighbor queries with UB-Trees and the Tetris algorithm for efficient access to a table in arbitrary sort order. We then describe the complexity of the involved algorithms and give solutions to selected algorithmic problems for a prototype implementation of UB-Trees on top of several RDBMSs. A cost model for sort operations with and without range restrictions is used both for analyzing our algorithms and for comparing UB-Trees with state-of-the-art query processing. Performance comparisons with traditional access methods practically confirm the theoretically expected superiority of UB-Trees and our algorithms over traditional access methods: Query processing in RDBMS is accelerated by several orders of magnitude, while the resource requirements in main memory space and disk space are substantially reduced. Benchmarks on some queries of the TPC-D benchmark as well as the data warehousing scenario of a fruit juice company illustrate the potential impact of our work on relational algebra, SQL, and commercial applications. The results of this thesis were developed by the author managing the MISTRAL project, a joint research and development project with SAP AG (Germany), Teijin Systems Technology Ltd. (Japan), NEC (Japan), Hitachi (Japan), Gesellschaft für Konsumforschung (Germany), and TransAction Software GmbH (Germany).

Acknowledgements

I thank my supervisor, Prof. Rudolf Bayer, Ph.D., for his support and the many fruitful discussions and inspirations. He helped me a lot with his ideas and his confidence, especially in the beginning, when the feasibility of the work and the road ahead were still unclear.

My master students and interns did a large portion of the prototype implementation. In addition they helped with analysis, and carried out many of the performance measurements. I especially thank Nils Frielinghaus, who was the first student who dared to be supervised by me. Our discussions produced ideas, which were crucial for the success of the MISTRAL project. The same holds for the work of Roland Pieringer, who helped to prove the practical benefits of our approach by porting our pilot implementation to Oracle and doing performance measurements at SAP in Walldorf.

When the MISTRAL project grew, I was not alone anymore: My team members Robert Fenk, Frank Ramsak, Stefan Sixl, and Martin Zirkel worked on the project with the same enthusiasm as I did. We had numerous inspiring discussions. For this thesis, my colleagues also did a lot of proofreading, which certainly improved the quality of the work.

For funding the research work I thank Karl-Heinz Hess and Klaus Majenz from SAP AG, Ichiro Arima, Shuichi Osaki and Toshishiro Satomi from Teijin Systems Technology, Japan, Kunitoshi Tsuruoka from NEC’s C&C Research Lab, Japan, Kazuo Masai and Shunichi Torii from Hitachi Ltd., Japan, Dr. Klaus Elhardt and Dr. Christian Roth from TransAction Software, and Dr. habil Thomas Ruf from GfK. I especially thank Klaus Majenz from SAP, Dr. Klaus Elhardt and Dr. habil Thomas Ruf for the discussions at our project meetings, which raised and answered many theoretical and practical questions. In addition, these discussions provided insight in database implementation and the practical demands of database users.

I also thank many further students, colleagues and friends for discussions, ideas and proofreading of the thesis. Here I especially thank Markus Blaschka, Bernd Deifel, Prof. Dr. Wolfgang Kowarschick, Peter Müller, Carsten Sapia, Michael Walther, and Wolfgang Wohner. Many thanks also to Prof. Johann Christoph Freytag, Ph.D., whose comments and ideas were a valuable contribution to my work.

Thanks also to my family and friends for helping me to forget my work, whenever I was stuck and needed some time to relax. I also thank Andrea for the 0.5l tea mug which was a good companion during some long nights in the office.

Last but not least I thank the Deutsche Bahn (German Rail), since the solution to many algorithmic problems (like the range query algorithm, multidimensional hierarchical clustering) and many new ideas (like the names MISTRAL and Tetris) were invented on the tracks.

Table of Contents

Part I

Preliminaries

Of course a film should have a beginning, a middle and an end. But not necessarily in that order.

(Jean-Luc Godard)

Chapter 1

Introduction

Multidimensional access methods are useful for a broad range of database applications, e.g., data warehousing, geographical information systems, data mining, archiving systems, lifecycle management databases. In general these access methods are beneficial, if queries define (and thus process) some part of a large data set by restrictions with respect to several categories (e.g., the sales for one year for a certain product, all cities near Munich with a population larger than 100.000 people). We focus on using multidimensional indexes for query processing in relational database management systems (DBMS). This introduction defines the scope and objective of the thesis. The applicability of multidimensional access methods for query processing is motivated by examples from data warehousing, non-standard DBMS applications and relational databases in general. The introduction also surveys related work in the fields of multidimensional access methods, relational DBMS and query processing, data warehousing and benchmarking. The chapter is concluded by giving a roadmap on the following chapters of the thesis.

1.1 Motivation

Complex business applications like SAP R/3 [SAP99], data warehousing (DW) and data mining as well as non-standard DBMS applications like geographical information systems (GIS) and statistical databases have created a strong demand for efficient processing of complex queries on huge databases. These complex queries set new requirements on the query processing and access method algorithms of a DBMS.

The main reason of indexing a table of a database is to accelerate query execution. This is achieved by utilizing the restrictions imposed by a query in order to reduce the number of disk accesses. We focus on using *multidimensional indexes* to organize any table of a database. In the following we use the terminology of relational DBMS because of their commercial importance. Since access methods are relevant to any DBMS regardless of the DBMS paradigm, the methodology described in this thesis can also be used to accelerate deductive DBMS (DDBMS, e.g., [Ull89, Spe91]), object-oriented DBMS (OODBMS, e.g., [BDK92]) or multidimensional DBMS (MDBMS, e.g., [PDF+98]).

We consider each tuple of a table to be a point in multidimensional space. Then a table describes a certain subspace of a multidimensional space defined by the cross product of the domains of all attributes. We use a multidimensional access method to organize that multidimensional space, which allows to cluster the data with respect to multiple attributes at the same time. As we will show, this multidimensional organization of a relational table is in many cases superior to one-dimensional clustering (i.e., organizing the data on disk for efficient access with respect to one attribute). However, current commercial DBMS do either not support multidimensional indexes at all or only use them as an add-on in the context of geo-spatial applications [Ora97, IBM97].

In the following we describe how data warehousing, non-standard DBMS as well as standard DBMS may benefit from the techniques described in this thesis.

1.1.1 Data Warehousing Applications

Data-warehousing applications [Inm96, Kim96, Dev97] cope with enormous amounts of data ranging in Gigabytes and Terabytes. While transactional (*OLTP, online transaction processing*) DBMS like bank applications usually use simple query patterns to retrieve a very small part of a database (usually one record) by a primary key access, data processing in data warehousing (*OLAP*, *online analytical processing*) involves complex queries that usually access a large portion of the database [CD97, WB97, GBL+96].

On the conceptual level a multidimensional (MD) view on the data models has been established by academia and the industry for OLAP applications [CD97]. In the MD model the numeric (quantitative) data (*measures*) (e.g., sales, cost), which is the focus of the analysis, is organized along multiple *dimensions*. The dimensions provide categorical (qualitative) data (e.g., container size of a product), which determines the context of the measures. Therefore the measures can be seen as a value in a multidimensional space – one often refers to this model as a multidimensional *cube*. An important concept of OLAP data models is the notion of *dimension hierarchies*. Hierarchies are used to provide structure to the otherwise flat dimensions. Often the data in the dimensions can be categorized/classified according to some additional characteristics (e.g., shops could be classified according to their location). Usually OLAP users are not interested in the single measures but in some form of summarized data (e.g., sales in a certain area). Hierarchies provide an appropriate method of describing the level of aggregation for a dimension.

Data processing in DW applications retrieves aggregated measures organized or classified according to several dimensions or hierarchies over the dimensions (e.g., the total sales for all coffee shops in Bavaria in 1999). For this reason multidimensional data models [BSH+98], multidimensional query languages (e.g., MDX [MS98b] or the OLAP Council approach [OLA98]) and even multidimensional DBMS (MDBMS) have been developed by the research community and implemented as commercial products. Typical OLAP operations are drill-down, roll-up and slice-and-dice [Kim96] and usually multiple dimensions are restricted at the same time. In general one can state that these operations in a MD model lead to range restrictions on the lowest hierarchy level of each dimension [Sar97].

To a large extent, relational DBMS are used for decision support applications, since these systems are well researched and are reported to provide more efficiency for huge databases than MDBMS. Regardless whether a multidimensional or relational paradigm is used to model and query OLAP data, queries result in multidimensional range restrictions in combination with sort operations and aggregations. Therefore any DBMS storing OLAP data must efficiently handle this typical query pattern.

Pre-computation, clustering and indexing are common techniques to speed up query processing. Pre-computation results in the best query response time at the expense of load performance and secondary storage space. For DW applications, pre-computation is mostly discussed for aggregation operations [CD97]. However, one requirement of DW is to efficiently deal with ad-hoc queries. Deciding which queries to pre-compute becomes extremely difficult then. Pre-computation also leads to a view maintenance problem.

In Section 8.2 we will illustrate the applicability of our technique to OLAP scenarios by using a multidimensional index to cluster the data cube of a data warehouse in order to efficiently process OLAP queries. We present a methodology to cluster data organized along multiple hierarchical dimensions. For clustering the fact table of a relational OLAP implementation we present performance measurements for a fruit juice company using a star schema with a fact table of 26 million records (an overall size of 7 GB). On this real-world data we experienced a performance increase up to a factor of ten compared to traditional techniques.

1.1.2 Non-Standard DBMS

Non-standard DBMS deal with complex objects which usually are not simply stored in tables or relations due to their interaction, relationship or size. Typical examples of non-standard DBMS are multimedia DBMS, CAD/CAM DBMS or geographical information systems (GIS). The systems usually aim towards specific applications, like similarity search, for example searching a picture or sound similar to an example picture/sound specified by a query. In many application areas like CAD [BKK97], computer vision [Jag91], multimedia archives [SH94], medical imaging [KSF+96], molecular biology [AGM+90] or time sequence analysis [AFS93], these problems can usually be reduced to search problems by feature transformation. However, often feature transformation results in high-dimensional data space which should be indexed with index structures specifically designed for these data spaces [Böh98, WSB98].

This thesis does not deal with high dimensional data spaces (see [Böh98] for a survey on indexing methods on high dimensional spaces). However, often proper data modeling allows to reduce dimensionality to below 10 dimensions. In this case, the methods described in this thesis may also be applied to non-standard DBMS applications. Moreover, since we aim to integrate our approach into standard DBMS, our technology may enable standard DBMS to deal with some subset of non-standard applications.

1.1.3 Relational DBMS and DBMS in General

Since tuples in a relation and points in multidimensional space are merely two different views on a data set, the data stored in databases can be considered to be of multidimensional nature. The expressiveness of query languages allows to write complex queries which mostly deal with more than one attribute. SQL queries, for instance, often involve multi-attribute restrictions and aggregations. These restrictions usually map onto ranges in several dimensions. In Section 2.1.4 we will show some SQL range queries against a database schema which is typical for decision support applications. Numeric attributes and time values are prime candidates to be restricted to ranges. Yet also restrictions in hierarchical data types will map onto ranges, if the data modeling of Section 5.3.4 is used. The index structures used in present commercial database systems are mostly based on B-Trees, which do not support multidimensional range queries efficiently. In Chapters 7 and 8 we will show the benefit of multidimensional index structures for processing complex queries with multi-attribute restrictions.

A sort operator utilizing multi-attribute restrictions is one of the most important operations for the physical layer of a DBMS, since sorting is required for an efficient implementation of most operations of the relational algebra (like for ordering a table, for grouping and aggregation, for projection as well as for sort-merge joins of several tables). In Section 4.7 we describe a single operator for efficiently processing sort operations with multi-attribute restrictions, the so-called Tetris algorithm [MB98]. The Tetris algorithm is a generalization of a multidimensional range query algorithm that efficiently combines sort operations with the evaluation of multi-attribute restrictions. The basic idea is to use the partial sort order imposed by a multidimensional partitioning in order to process a table in some total sort order. With the Tetris algorithm a multidimensional index can reduce resource requirements for virtually any operation of the relational algebra. Compared to the native access methods of a commercial DBMS, our prototype implementation of the Tetris algorithm shows significant speedups for queries of the TPC-D benchmark. In addition, temporary storage requirements for the sorting process are reduced and first results of a sort operation are available much earlier for further processing. In Section 6.4 we will see how multidimensional index structures and the Tetris algorithm can be used to accelerate virtually any operation of the relational algebra.

Sort operations and multi-attribute restrictions are not only useful for RDBMS implementation, but for DBMS in general. Query processing in OODBMS, DDBMS or MDBMS also relies on efficient access to data in some sort order (e.g., for duplicate elimination or set operations). Thus the results of this thesis are applicable to any DBMS.

1.2 Objective

The goal of this work is to show the potential of integrating multidimensional indexes as first class indexes into the kernel of a database system. We intend to provide a deeper insight into the problems and chances of multidimensional indexing. Therefore this thesis includes two analytical chapters: Chapter 3 studies multidimensional space partitioning and Chapter 6 derives a cost model for range queries with and without sort operations. This cost model is further used to analyze the range query performance in order to have a benchmark to judge our practical measurements.

Another objective is to show the practical feasibility of our approach. We therefore describe the main challenges of our prototype implementation in Chapter 4 and give real performance figures for both artificial and real-world data in Chapters 6 and 7. In order to illustrate further work in this area and give hints for further reading we survey related work in Section 1.3.

In addition we aim at introducing two new technologies, which were developed by the author during his work on the thesis, namely:

- the Tetris algorithm for processing queries with multi-attribute restrictions and sort operations (see Section 4.7)

- multidimensional hierarchical clustering for clustering data organized according to multiple hierarchical dimensions (see Section 5.3.4)

These techniques may significantly speed up query processing and DBMS and therefore may be of great commercial value. While the Tetris algorithm is a technique which extends the already patented UB-cache idea [Bay97b], a patent application for multidimensional hierarchical clustering is planned by the author and his supervisor.

Many literature has been published about surveying or comparing multidimensional access methods. We just briefly describe the main papers and main approaches in the next section. Instead, our focus is to study the impact of multidimensional index structures on relational query processing. For performance comparisons of multidimensional indexes we also refer to the related work listed in Section 1.3.

1.3 Related Work

In the following we survey related work in the field of multidimensional access methods, relational DBMS and query processing, which may be used for reference or further reading.

1.3.1 RDBMS and Query Processing

An important task of query processing in RDBMS is to efficiently implement algorithms for the basic operations of the relational algebra [Cod70]. Usually, these algorithms apply to particular storage structures or access methods. [Gra93] gives a concise survey of query processing. The selection operation is either implemented by a table scan, or, if an index is available, by an index scan.

Indexing is used to efficiently process a query if the result set defined by the query restrictions is fairly small. Most OLTP applications use B-Trees [BM72, Com79] as their standard indexing scheme. For point-restrictions it is also possible to use hash indexes [FNP+79,].Favoring retrieval response time over update response time allows to build several indexes on one table or data cube of a DW. Bitmap indexes (e.g., [OQ97, CI98, WB98]) are widely discussed as an improvement over B-Trees for DW applications, since they efficiently evaluate queries with multi-attribute restrictions. However, the overall result set still must be relatively small. This is a major drawback of bitmap indexes, since usually a relatively large part of a cube must be accessed in order to calculate aggregated measures.

Clustering places data that is likely to be accessed together physically close to each other. The goal of clustering is to limit the number of disk accesses required to process a query by increasing the likelihood that query results have already been cached. Clustering has been well researched in the field of file structures and access methods (e.g., [Sal88, GR97, Sto94]. B-Trees, for instance, provide one-dimensional clustering. Multidimensional clustering has been discussed in the field of multidimensional access methods (e.g., [GG97, Sam90]

A great deal of research has been done in the field of access methods and access path selection. Especially for DW environments, specialized access methods have been proposed [Inf97, OQ97]. Often several indexes are created on one table in order to speed up query processing [Lum70, Mul71, MHW+90, Red97, GHR+97].

If the selection condition specifies a range in a single attribute, a clustering index greatly speeds up query processing. Conjunctive selection conditions are efficiently processed by composite indexes, intersection of record pointers or multidimensional indexes. We will investigate processing of multi-attribute restrictions in more detail in Section 2.4 and Chapter 6.

One of the most important operations in RDBMS is the join operation, which is used to combine several relation which were normalized during data modeling. The join operation is usually implemented by nested loop algorithms, join indexes, sort-merge algorithms, or hash algorithms. [ME92] surveys join processing in relational databases.

The relational operations of projection, union, intersection and set difference are efficiently implemented by processing a relation in some sort order and then either use an index scan or merge-sort algorithm [Gra93]. Efficient sort operations and the use of restrictions to limit result sets are crucial to many query processing algorithms. Very often queries combine several operations of the relational algebra like join and restriction.

1.3.2 Multidimensional Access Methods

Multidimensional access methods are well researched in the field of spatial databases. [GG97] and [Sam90] provide excellent surveys of almost all of these methods. Multidimensional indexes are used to utilize spatial restrictions (e.g., range restrictions, intersection, overlap) and to efficiently compute spatial joins [Rot91, Gün93, BKS93]. Multidimensional data structures are usually classified as *point access methods* storing points in multidimensional space and *spatial access methods* storing multidimensional extended objects of arbitrary volume, e.g., boxes, spheres, etc. Usually one also distinguishes between main memory structures, which are used to manage multidimensional data in main memory, and secondary storage structures, which are used for efficiently accessing large multidimensional databases on secondary storage.

Thus, a lot of effort has been put into indexing spatial data. It is argued that these data structures can also be used to index point data. However, point data has special properties that should be exploited by a multidimensional index. The most important property is that the multidimensional space can be partitioned into disjoint subspaces without introducing redundancy when storing objects. Thus redundancy considerations as required for multidimensional extended objects [Ore89] are not necessary. A disjoint partitioning of a multidimensional space is very desirable since it enables to give logarithmic performance guarantees for insertion, deletion and exact match queries [Bay96]. Furthermore, any relational table in a RDBMS stores multidimensional point data. Therefore this type of data is of great commercial interest. So multidimensional indexes for point data are useful in a large market segment. Therefore in this thesis we focus on indexing multidimensional point data.

[GG97] distinguishes three categories of multidimensional point access methods: Techniques based on *hashing* (grid files [NHS84], EXCELL [Tam82], multi-level grid files [Hin85], twin grid files [HSW88b] and multidimensional hashing [Fal85, Fal88]), *hierarchical access methods* (K-D-B-Tree [Rob81], LSD-Tree [HSW89], Buddy Tree [SK90], BANG File

[Fre87], hB-Tree [LS90], R-Trees [Gut84, SRF87, BKS+90, BKK96]) and *space filling curves in combination with one-dimensional access methods* [TH81, OM84, Jag90, AS83, FR89]. Yet another access technique is to use a *combination of several one-dimensional access* methods like inverted files [Lum70, MHW+90] or bitmap index intersection [OQ97].

We just briefly sketch the problems of the main approaches here. A detailed description is found in the original papers, a detailed comparison is found in [GG97].

Grid-files give a two-access-guarantee for retrieval, but have an extremely bad worst-case behavior for updates: Inserting a point may result in a non-local split of the grid and thus require a reorganization of the grid-file. Furthermore, grid files have problems with dependencies in the multidimensional data distribution. For linearly dependent data the grid may require more storage than the tuples stored in the grid.

k-d-B-Trees exhibit a forced split effect, which does not allow one to give any space utilization guarantees. In worst case a large amount of pages may be completely empty.

hB-Trees have a complex organization and extremely difficult algorithms, since they are a hybrid data structure. In addition hB-Trees may store several references of a node to the same child node, which may result in a superlinear growth of the index nodes with respect to the number of regions in space.

R-Trees cannot give any performance guarantee for the basic operations, since they do not partition the multidimensional space in disjoint parts, but allow overlapping rectangles. Successors of the R-Tree like the R*-Tree [BKS+90] and the X-Tree [BKK96] use complicated algorithms or even introduce buckets of varying size to minimize overlaps. However, complicated algorithms cannot overcome this problem in general. Introducing buckets of varying size may cause the index to degenerate. So the basic problem of R-Trees still remains.

Yet a very promising approach to store multidimensional point data is to map the data onto a one-dimensional space filling curve [Sag94] like the Z-curve or the Hilbert curve and use the properties of this curve for efficient retrieval. The biggest advantage of space-filling curves over the techniques described before is that they allow a disjoint partitioning of the multidimensional space. In addition the storage requirements do not degenerate for any data distribution. Well known one-dimensional indexing methods can be applied and multidimensional search problems are reduced to linear search problems. Hence multidimensional insertion, deletion and point query algorithms inherit the complexities of the corresponding one-dimensional access method. Using B-Trees as one-dimensional access method allows to give logarithmic performance guarantees for the basic operations of insertion, deletion and point queries.

Our approach also relies on space filling curves. Our pilot implementation has some similarity to the zkd-B-Tree approach described in [OM84]. We also use the Z-curve to to transform multidimensional point data into one-dimensional data. In contrast to zkd-B-Trees, we do not store tuples in Z-representation, but only use Z-addresses to define a partitioning concept for the multidimensional space. This yields a better space partitioning: Our approach has more freedom of choice to pick a suitable split-point for the partitioning. In addition it allows more efficient tuple extraction algorithms, since it is not necessary to store the Z-representation of every tuple.

The major problem of multidimensional index methods in general is the curse of dimensionality (e.g., [WSB98]), since the number of possible partitionings explodes exponentially with the number of dimensions. This dimensional curse forbids clustering of high-dimensional data. However, often problems that seem high-dimensional at first glance have a reduced dimensionality, if proper data modeling is used. In data warehousing applications one seldom has more then 10 independent dimensions, usually even much fewer. If a data warehouse utilizes more dimensions, there are usually some dependencies between the dimensions. Customers, for example, tend to buy the same product. In the same way, many customers exist only for a certain time. If a data structure is to organize the multidimensional space, it would be beneficial if those dependencies are taken into account in order to limit the number of partitions.

1.4 Outline

This thesis is divided into three parts. Part One consists of the first three chapters and describes preliminaries of our work. The second part describes our approach to relational query processing with multidimensional indexes. In the third part we investigate our technique of Part Two both theoretically by a cost analysis and practically by examples and performance measurements. In the following we briefly sketch the contents of each chapter.

Chapter 2 gives basic definitions which will be used throughout the thesis. In addition we provide an overview on query processing concepts with a special focus on multidimensional range queries.

Chapter 3 gives a detailed analysis of multidimensional space partitioning. We identify space filling curves to create a linear ordering of a multidimensional space. Relying on one special kind of space filling curve, namely the Z-curve, we then introduce the concept of Z-regions to define a partitioning of the multidimensional space. We investigate and prove several important properties of Z-regions, especially their local proximity and connection in space. We also investigate the limitations of our approach to multidimensional space partitioning with respect to data distributions and increasing dimensionality.

The UB-Tree, a multidimensional index based on space filling curves and B-Trees, is described in Chapter 4 together with its basic algorithms for insertion, deletion, point queries and range queries. In addition we introduce two new algorithms for UB-Trees in this chapter, namely the *spiral algorithm* for the evaluation of nearest neighbor queries and the *Tetris algorithm* for processing sort-operations with multi-attribute restrictions.

Chapter 5 describes selected algorithmic problems of our prototype implementation on top of several commercial RDBMS. In this chapter we also investigate how to deal with various data types and data distributions. We especially consider how to organize a multidimensional space, whose dimensions can be organized hierarchically. This concept of multidimensional hierarchical clustering is especially applicable to data warehousing applications. In Chapter 8 it is used to cluster the fact table of a relational data warehouse.

In Chapter 6 we derive a cost function for multi-attribute restrictions in multidimensionally partitioned universes. We further define a cost model and cost functions for the techniques that prevail in current RDBMS for the processing of multi-attribute restrictions. We then analyze the performance of the range query algorithm and the Tetris algorithm for UB-Trees and compare it to other access methods that prevail in present RDBMS.

Performance measurements in a laboratory environment with generated, uniformly distributed test data are presented in Chapter 7. Although we mainly investigate the range query performance, we also briefly sketch point queries and insertion there. This chapter is more of analytical interest and is aimed at providing a better practical understanding of the effects of multidimensional clustering. In addition, by actual performance measurements it undermines the correctness of the results that were derived in Chapter 6 using our cost model.

Chapter 8 describes the impact of our approach on query processing. We list transformation rules which may be used to implement the basic relational operators (selection, projection, ordering, grouping and aggregation, equi-join, set operations) with UB-Trees. We used these transformation rules to apply our multidimensional query processing techniques in two real world applications scenarios, the TPC-D benchmark and a star schema data warehouse. Performance measurements and comparisons for the TPC-D schema with 2 GB of generated data and the data warehouse with 7 GB of real world data are also presented in that chapter.

Chapter 9 concludes the thesis with a summary and an outlook on future work.

1.5 How to Read the Thesis

We tried to use well-accepted technical terms whenever possible in order to avoid confusion with other research work in the field of multidimensional indexes and RDBMS. Thus experienced readers may skip most of Chapter 2, which essentially gives basic mathematical definitions as well as a quick overview of the state of the art in query processing and defines a basic terminology. However, in order to fully judge our approach of multidimensional clustering we recommend to read Section 2.3 in any case.

Chapter 3 is mainly relevant to the mathematically interested reader, who wants to get a deeper insight into the field of space filling curves and their relevance to multidimensional data processing. It is also helpful to judge the chances and limitations of multidimensional indexing.

Chapter 4 is to understand how UB-Trees and query processing with UB-Trees work. UB-Trees are defined via space filling curves in this thesis. Therefore it might be helpful to have read the sections 3.1 and 3.4 before reading Chapter 4.

Chapter 5 describes some pitfalls that occurred when implementing a prototype of the UB-Tree access method on top of several RDBMS. This chapter might also be useful to anyone wanting to apply UB-Trees for a specific database schema, since it also introduces the concept of transformation functions, which in the case of variable UB-Trees and multidimensional hierarchical clustering has relevance to physical data modeling.

In Chapter 6 the mathematically interested reader finds a formal treatment of range query performance. Readers with a special focus on the field of query processing and query optimization might also have a look at this chapter, since it defines and analyzes a cost model for UB-Trees as well as other access structures,

The performance measurements of Chapters 7 and 8 are of interest to anybody wanting to evaluate the quality of our approach. Chapter 7 is to undermine the correctness of our cost model laid out in Chapter 6. The performance figures and query processing proposals of Chapter 8 are especially relevant to practitioners in the field of RDBMS development and RBDMS applications.

Every science, like a recurring decimal, has a beginning and no end.

(Anton Chekov)

Chapter 2

Terminology and Basic Concepts

In relational database schemas, tables often bear composite primary keys concatenated of several attributes. For efficient query processing this composite key is used as primary index to physically organize the table on secondary storage. In the same way, secondary indexes for a table often consist of a concatenation of attributes. This thesis investigates the usability of a multidimensional access method for multi-attribute keys. This chapter introduces the duality of points in multidimensional space and tuples of a relation. We explain our terminology and give basic definitions which will be used in the following chapters of this thesis. We also provide a classification for query types common in today's database applications. We address the problem of indexing in general and look on range queries more closely.

2.1 The Multidimensional Space: Duality of Points and Tuples

Relational database management systems (RDBMS) have a large market share for commercial applications. Although the techniques described in this thesis are very well applicable to speed up database management systems in general (e.g., hierarchical databases, CODASYL databases, or object oriented databases, see e.g., [Ull88]), we use the terminology of RDBMS in the following. In the relational world data is stored in a set of *relations*. We use the term *table* synonymously to relation. Each table is organized into *rows* and *columns*. We use the terms *tuple* synonymously to row and *attribute* synonymously to column. The number and type of the attributes is fixed for each relation; thus each tuple has the same number of typed attributes. We call this number of attributes *arity* of a relation. The *type* of each attribute represents the set of permissible values called *domain*.

In this thesis we consider a tuple to be a *point* in multidimensional space. We use the term *universe* to denote the multidimensional space defined by the domains of the attributes. Each attribute determines one *dimension*. The value of an attribute of a tuple is therefore the *coordinate* of a point in multidimensional space. The duality of points and tuples causes the following terms to be used synonymously in this thesis:

- (multidimensional) domain, universe, multidimensional space
- relation, table, subset of multidimensional space
- row, tuple, point
- attribute, column, coordinate, dimension
- arity, dimensionality

In general we write $|S|$ to denote the cardinality of any set S. For any string s we write $|s|$ to denote the length of s.

2.1.1 Relations, Tuples, Attributes and the Multidimensional Space

Let R be a relation having d attributes $A_1,..., A_d$ of domains $\mathbb{A}_1,..., \mathbb{A}_d$. R is a set of tuples $x = (x_1,..., x_d)$. Let $<_i$ be a total order on $\mathbb{A}_i$ and λ_i resp. υ_i the minimum resp. maximum value of $\mathbb{A}_i$. We call d the arity of R. $|R|$ is the cardinality of R, i.e., the number of tuples stored in R.

When writing tuple literals we sometimes omit the brackets and commas. For instance, we write “ab” as abbreviation for the tuple literal “(a, b)”.

For notational convenience we define the set of dimension indices as $D =\{1, ..., d\}$.

To simplify theoretical analysis, we consider the domain $\mathbb{A}_i$ of A_i, $i \in D$ to be mapped to Ω_i, a set of non-negative integer numbers. Thus, for each value x_i of A_i we get:

$$x_i \in \Omega_i = \{0, ..., r_i\text{-}1\} \subset \mathbb{N}_0$$

Thus for Ω_i we get $\lambda_i = 0$ and $\upsilon_i = r_i\text{-}1$.

In addition we require $r_i = 2^v$ for some arbitrary $v \in \mathbb{N}_0$. This is not a restriction, since every finite[1] totally ordered domain $(\mathbb{A}_i = \{a_1,\ldots,a_r\} , <_i)$ with $r = |\mathbb{A}_i| \leq r_i$ can be mapped monotonically to Ω_i by :

$$f\colon \mathbb{A}_i \rightarrow \Omega_i, \text{ so that } f(a_j) < f(a_k) \Leftrightarrow a_j <_i a_k$$

Definition 2-1 (multidimensional domain, Ω): The *multidimensional domain* Ω of the entire relation is the cross product

$$\Omega = \Omega_1 \times \ldots \times \Omega_d = \{0,\ldots,r_1\text{-}1\} \times\ldots \times \{0,\ldots,r_d\text{-}1\}.$$

We call Ω the *base space* of R. Thus R is a finite subset of Ω, i.e., $R \subseteq \Omega$

The cardinality of Ω then can be calculated as:

$$|\Omega| = \prod_{i=1}^{d} r_i$$

Definition 2-2 (⊴-order of Ω): For the multidimensional space Ω we define a partial order: For $x, y \in \Omega$

$$x \trianglelefteq y \Leftrightarrow x_i \leq y_i \text{ for all } i \in D$$

$$x \triangleleft y \Leftrightarrow x_i < y_i \text{ for all } i \in D$$

In the following we use the symbol $\mathbb{D}$ to denote a domain of values.

Definition 2-3 (<-neighbors): For any ordering relation < and an ordered domain $(\mathbb{D}, <)$ two values $a, b \in \mathbb{D}$ are *<-neighbors*, if and only if, $a < b$ and there exists no $c \in \mathbb{D}$ with $a < c < b$. For $a \in \mathbb{D}$ define:

<-neighbors(a) = $\{b \in \mathbb{D} \mid b$ and a are <-neighbors or a and b are <-neighbors$\}$

If the ordering relation < is obvious, we just write neighbors(a) instead of <-neighbors(a).

Lemma 2-1 (◁−neighbors): Two points $x, y \in \Omega$ with $x \triangleleft y$ are ◁−neighbors, if and only if, there exists an index i so that $x_j = y_j$ for all $j \in D\backslash\{i\}$ and x_i and y_i are <-neighbors.

Proof:

The proof is a direct consequence of Definition 2-2 and Definition 2-3. □

[1] Theoretically, the approach described in this thesis could also operate on infinite domains. Then one only needs a split function which partitions an interval of the domain into two disjoint intervals (see in Section 3.6). However, it suffices to consider finite domains, since in a computer every domain like real numbers, integer numbers, character strings, etc. is represented by a finite domain.

Lemma 2-2: A point x in d-dimensional space has at most $2 \cdot d$ $\lhd$−neighbors.

Proof:

The proof is a direct consequence of Lemma 2-1. □

Definition 2-4 (distance of two values): For any totally ordered set ($\mathbb{D}$, <) and $a, b \in \mathbb{D}$ with $a < b$ we define

$$\text{distance}(a,b,<) = \begin{cases} \left| \{ c \mid a < c < b \} \right| + 1, & a \neq b \\ 0 & , a = b \end{cases}$$

We often are not interested in the specific domains of the attributes. For an easy mathematical treatment we therefore define ^, a normalization operation of each domain to a value of the interval [0, 1].

Definition 2-5 (scalar normalization): If a is a value in a domain $\mathbb{D} = [a_{\min}, a_{\max}] \subset \mathbb{N}_0$, then we normalize in the following way:

$$\hat{a} := \frac{a - a_{\min} + 1}{a_{\max} - a_{\min} + 1}$$

For any attribute value $x_i \in \Omega_i = \{0,...,r_i\text{-}1\}$, $i \in D$, of A_i of a tuple $x \in \Omega$ we get:

$$\hat{x}_i := \frac{x_i + 1}{r_i}$$

Definition 2-6 (tuple normalization): For a tuple $x \in \Omega$ we define $\hat{x} := (\hat{x}_1,...,\hat{x}_d)$.

2.1.2 Intervals

Definition 2-7 (one-dimensional intervals): For any totally ordered domain ($\mathbb{D}$, <) and a pair of values $a, b \in \mathbb{D}$ with $a < b$ we define the one-dimensional interval:

$$[a, b] = \{c \in \mathbb{D} \mid a \leq c \leq b\}$$

If the point describing the lower bound (or the upper bound or both bounds) is not included in the interval, we write:

$$]a, b] = \{c \in \mathbb{D} \mid a < c \leq b\}$$

$$[a, b[= \{c \in \mathbb{D} \mid a \leq c < b\}$$

$$]a, b[= \{c \in \mathbb{D} \mid a < c < b\}$$

Definition 2-8 (multi-dimensional interval): For two points $y, z \in \Omega$ with $y \trianglelefteq z$ we extend Definition 2-7 to multidimensional intervals:

$$[[y, z]] = [y_1, z_1] \times ... \times [y_d, z_d] = \{(x_1,...,x_d) \mid y_i \leq x_i \leq z_i \text{ for all } i \in D\}.$$

We define $]]y, z]]$, $[[y, z[[$ and $]]y, z[[$ analogously.

Note that multidimensional intervals are iso-oriented with respect to all dimensions, i.e., their faces are parallel to the coordinate axes. Because of their rectangular nature we also use the term *box* to denote a multidimensional interval.

2.1.3 Volumes

We use the term *volume* for all dimensionalities instead of using the terms length for linear spaces, area for two-dimensional spaces, volume for three-dimensional spaces or Jordan-content for higher dimensional spaces. For simplicity by volume of a part of a space we mean the normalized volume with respect to the entire space. Because of our discrete model we consider a single value of a domain $\mathbb{D}$ to have the volume $1/|\mathbb{D}|$.

Definition 2-9 (volume of a linear interval): For a linear (one-dimensional) interval $[x_\mathrm{i}, y_\mathrm{i}] \subseteq \Omega_i = \{0,...,r_i\text{-}1\}$ we define its volume:

$$\mathrm{vol}([x_i, y_i]) = \hat{y}_i - \hat{x}_i + \frac{1}{r_i}$$

The volume of the empty set is zero:

$$\mathrm{vol}(\varnothing) = 0$$

Thus a volume is a normalized number between 0 and 1. We generalize this one-dimensional volume definition to multi-dimensional intervals $[[x, y]]$:

Definition 2-10 (volume of a multidimensional interval):

$$\mathrm{vol}([[x, y]]) = \prod_{i=1}^{d} \mathrm{vol}([x_i, y_i])$$

Definition 2-11 (volume of a set of multidimensional intervals): We define the volume of a union of k disjoint multidimensional intervals S_j, $j \in \{1 ,..., k\}$, as the sum of the volumes of each interval:

$$\mathrm{vol}(\bigcup_{j=1}^{k} S_j) = \sum_{j=1}^{k} \mathrm{vol}(S_j)$$

Since each finite multi-dimensional point set can be decomposed into disjoint multi-dimensional intervals, the volume of any finite multidimensional point-set can be calculated by this formula.

Lemma 2-3: The volume of the difference of two point-sets S and $Q \subseteq S$ is the difference of the volumes of S and Q, i.e.,

$$\mathrm{vol}(S \backslash Q) = \mathrm{vol}(S) - \mathrm{vol}(Q).$$

Proof:

S can be decomposed into a set of disjoint multi-dimensional intervals $\{S_1,...,S_k, Q_1,...,Q_j\}$, such that $Q = Q_1 \cup ... \cup Q_j$. Then, $\mathrm{vol}(S) = \mathrm{vol}(Q_1,...,Q_j) + \mathrm{vol}(S_1,...,S_k) = \mathrm{vol}(Q) + \mathrm{vol}(S \backslash Q)$.

□

2.1.4 Statistical Functions

Definition 2-12 (average): For any set S of numbers we define its *average*:

$$\mathrm{avg}(S) = \frac{1}{|S|} \cdot \sum_{a \in S} a$$

Definition 2-13 (standard deviation): For any set of numbers S we define its *standard deviation*:

$$\mathrm{std}(S) = \sqrt{\mathrm{avg}\left(\left\{(s - \mathrm{avg}(S))^2 \mid s \in S\right\}\right)} = \sqrt{\frac{1}{|S|} \cdot \sum_{a \in S} (a - \mathrm{avg}(S))^2}$$

2.1.5 Partitioned Relations

In the following we define the terms page and region. We use these terms to denote the partitioning of a relation with respect to the partitioning of the corresponding base space. Relations are physically stored on pages of the secondary storage. A *page* is a physical unit of secondary storage that stores a certain capacity of tuples of a relation. A *region* is a subspace of the multidimensional base space of the relation. Because of the duality of points and tuples, a *set of regions* that partitions the base space Ω *corresponds* to a partitioning of the relation R into a *set of pages*.

Definition 2-14 (region):A *region* is a subspace of Ω. Regions are neither required to be rectangular nor connected. We write ρ_1, ρ_2, ρ_3.,... for regions.

Definition 2-15 (page; page capacity): A *page* is a fixed size byte container to store tuples of a relation. We write p_1, p_2, p_3,... for pages. The capacity of a page is the maximum number of tuples (or bytes for variable length tuples) that a page may hold. We denote the page capacity by C.

Definition 2-16 (correspondence between pages and regions): A *page p corresponds to a region ρ* ($p \leftrightarrow \rho$), if all tuples stored on p are located in the region ρ, i.e.,

$$p \leftrightarrow \rho \Leftrightarrow (x \in p \Leftrightarrow x \in \rho \cap p)$$

To physically store a relation R in a DBMS, R is partitioned into $P_R = \{p_1, ..., p_k\}$, a finite set of disjoint pages. Each page p_i, $i \in \{1, ..., k\}$ stores a limited number of tuples.

Definition 2-17 (region partitioning): A *region partitioning* of Ω for a partitioned relation $P_R = \{p_1, p_2, ..., p_k\}$ is a set of regions $\Theta = \{\rho_1, ..., \rho_k\}$ with

$$\bigcup_{i=1}^{k} \rho_i = \Omega \text{ and } \forall_{j,i=1,...,k \text{ and } j \neq i}\, \rho_i \cap \rho_j = \varnothing \text{ and } \forall_{i=1,...,k}\, p_i \leftrightarrow \rho_i$$

For B-Trees as used in standard RDBMS the region partitioning usually takes place with respect to one attribute or with respect to several attributes in some lexicographic order. Our region partitioning is more general and will be used in Chapter 3 to define a multidimensional partitioning of a relation.

2.2 Query Types

Relational queries are expressed by operators of the relational algebra and either deal with a single table or combine multiple tables [Cod70]. Single table queries restrict, re-arrange or aggregate the tuples of one relation [Ull88].

When asking a restriction query (denoted by the relational operator σ) we are interested in a subspace of a multidimensional universe. Depending on the shape and the volume of the query space we can distinguish different types of restriction queries. A large set of these queries can be reduced to partial range queries.

Re-arranging means

- sorting (denoted by ω),
- projecting (denoted by π) or
- grouping (denoted by γ) and aggregating (denoted by an aggregation function like sum or count)

the tuples of a relation.

The most frequent operation to combine multiple tables is the join operation (denoted by $\bowtie$), mostly the natural join. In this thesis we will present a new processing technique for the operations illustrated in Figure 2-1.

Definition 2-18 (query, result set): A *query* is a predicate $\varphi(x)$ over the tuples x of a relation R. The *result set* RS of a query is the subset of tuples stored in R satisfying the query predicate:

$$\mathrm{RS}(R,\varphi) = \{x \in R \mid \varphi(x)\}.$$

The *result set size* is the cardinality of the result set $|\mathrm{RS}(R, \varphi)|$.

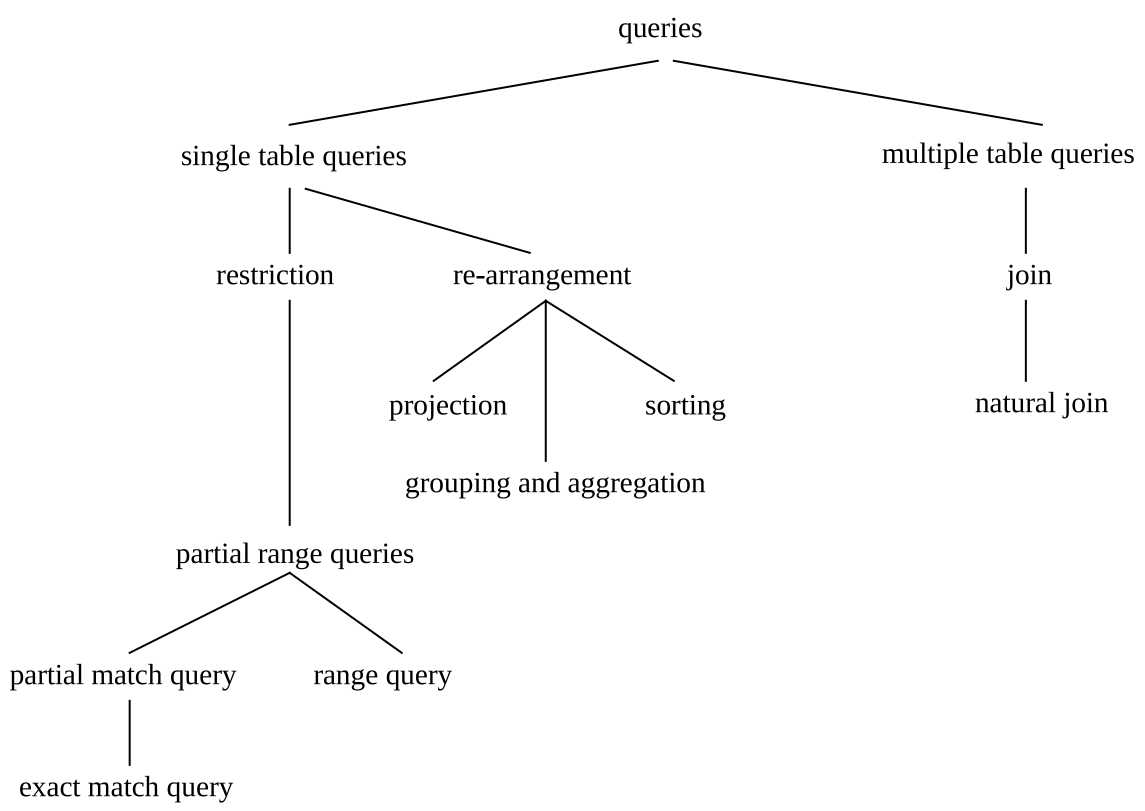


Figure 2-1: Query Categories

With the duality of multidimensional points and tuples, every query predicate $\varphi(x)$ defines a query space $\varphi(x) = Q \subseteq \Omega$. The result set is the set of tuples of the relation that is located in Q, i.e., RS = $\{x \in R \mid x \in Q\}$. The result set size is the number of tuples of R located in Q.

Definition 2-19 (selectivity): The *selectivity* of a query φ is the number of tuples in the result set of a query compared to the number of tuples stored in the relation:

$$\text{selectivity}(R, \varphi) = \frac{|\text{RS}(R, \varphi)|}{|R|}$$

Definition 2-20 (restriction): The query space of a query Q is defined by a *restriction* in none, some or all attributes. Typical restrictions are point restrictions, interval restrictions, restrictions depending on another attribute or restrictions depending on the data distribution in the database.

We mostly consider independent restrictions, i.e., the restriction in one attribute does not depend on the restrictions of the other attributes or the data distribution of the relation (see Section 3.8 for a treatment of dependent dimensions). Independent restrictions, for instance, are point restrictions and interval restrictions, if the points or intervals for each dimension are independent. We model queries with independent restrictions by an interval in each attribute. If an attribute is not restricted, we use the interval $[-\infty, \infty]$ resp. $[\lambda_i, \upsilon_i]$. A point restriction in an attribute is considered to be an interval with identical lower and upper bounds.

Definition 2-21 (restriction interval, query box): A *restriction interval* is a one-dimensional interval defining the restriction of one attribute in a query. A *query box* Q is a multidimensional interval defined by the restriction intervals of a query, i.e.,

$$Q = [[y, z]] = [y_1, z_1] \times ... \times [y_j, z_j] \times ... \times [y_d, z_d]$$

Without loss of generality we will use normalized restriction intervals for an easier mathematical treatment, i.e., both the lower bound and the upper bound of the restriction interval are values between 0 and 1. In the following we will often identify a restriction by the volume of the corresponding restriction interval.

Definition 2-22 (index candidate, index attribute, result attribute): We call an attribute x_i of a relation R to be an *index candidate* of a query $\varphi(x)$ over R, if x_i is restricted and/or ordered during the query processing of φ. We call an index candidate *index attribute*, if it is actually indexed by some index I (cf. Section 2.3) on R. We call an attribute of R *result attribute*, if it is not restricted or ordered when processing φ.

In this thesis the terms *index candidate* and *index attribute* denote physical concepts of a data model (which are used to derive secondary storage structures like indexing, partitioning, clustering or query materialization). In contrast to that the terms *candidate key* and *key attribute* as used in relational data modeling (e.g., [Ull88]) denote logical concepts of a data model.

Although result attributes are of great practical relevance, apart from increasing the size of the tuple they often[2] do not influence the behavior of a multidimensional index. Typically result attributes are projected (without duplicate elimination) or aggregated during processing a query. For an easier notation from now on we will omit result attributes of a tuple whenever possible. To be more specific, we will consider d (or d_R, when talking about several relations) to be the number of the index candidates of a relation R. If a relation has additional result attributes, we denote the arity of a relation by d' (or d'_R). The result attributes are $x_{d+1},...,x_{d'}$.

Note that the terms index candidate and result attribute are query dependent. For some query an attribute might be an index candidate, while it is a result attribute in another query. However, for many application scenarios like data warehousing the set of index candidates/attributes and the set of result attributes are constant and both sets are disjoint over a quite large set of typical queries.

We use the star-schema of the TPC-D benchmark [TPCD97] to illustrate different categories of queries by examples. For the examples of this section we use the ORDER and LINEITEM relations. Figure 2-2 shows the entire schema of the TPC-D benchmark. The value SF is the scaling factor and determines the number of tuples of each relation. The TPC-D benchmark was mainly developed to evaluate the capabilities of RDBMS for complex queries which are frequently found in decision support applications. Next to the schema and data distribution definition the specification consists of 17 pre-defined queries and two update functions.

[2] Besides increasing the tuple size result attributes are decisive for the performance of non-clustering indexes, since these attributes are not stored in the index but require one additional random access to the data file. This is analyzed in detail in Section 2.3.2.

For illustration purposes we define some additional queries in the next sections. We assume that O_TOTALPRICE, O_ORDERDATE, L_SHIPDATE, L_DISCOUNT and L_QUANTITY are index attributes which are used for the multidimensional organization of the corresponding table. Thus $d_{\text{LINEITEM}} = 3$, $d'_{\text{LINEITEM}} = 16$, $d_{\text{ORDER}} = 2$ and $d'_{\text{ORDER}} = 9$.We state each query both in a text version and in an SQL-version.

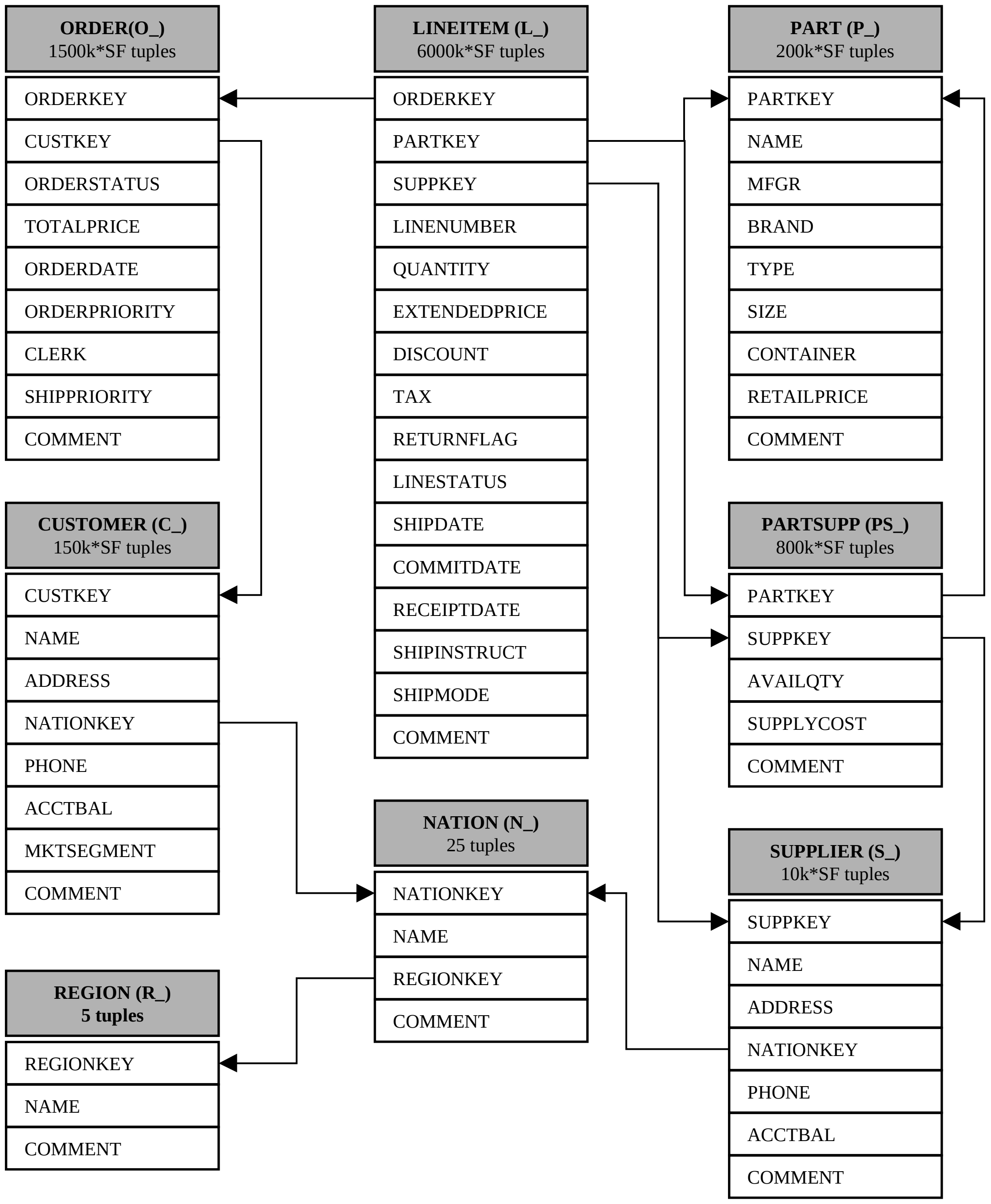


Figure 2-2: Schema of the TPC-D Benchmark

2.2.1 Partial Match Query

A *partial match query* restricts some dimensions to a point, while other dimensions are left unspecified. Formally, for a given set of indices $S \subseteq D$ and a point $p \in \Omega$ the result set of the partial match query PM(p, S, R) is:

$$\mathrm{PM}(p, S, R) = \{x \in R \mid x_i = p_i \text{ for all } i \in S\}$$

Example 2-1a:

“list all orders of 23.12.1997”
(Figure 2-3a)

```
SELECT O_ORDERKEY
   FROM ORDER
WHERE
   O_ORDERDATE = ″23.12.1997″
```

Example 2-1b:

“list all parts shipped on 23.12.1997”
(Figure 2-3b)

```
SELECT L_PARTKEY
   FROM LINEITEM
WHERE
   L_SHIPDATE = ″23.12.1997″
```

A special form of a partial match query is an *exact match query* (also called *point query*) that restricts all dimensions to a point. The result set EM(p, R) of an exact match query defined by a multidimensional point $p \in \Omega$ is:

$$\mathrm{EM}(p, R) = \{x \in R \mid x_i = p_i \text{ for all } i \in D\} = \mathrm{PM}(p, D, R)$$

Exact match queries are used to retrieve the result attributes of a point defined by equality restriction in the index attributes. These queries are often used for existence checking or referential integrity checking.

Example 2-1c:

“list all orders of 23.12.1997 with a total price of 10000” (Figure 2-3c)

```
SELECT O_ORDERKEY
   FROM ORDER
WHERE
   O_ORDERDATE = ″23.12.1997″ AND
   O_TOTALPRICE = 10000
```

Example 2-1d:

“list all parts that have been shipped on 23.12.1997 with a quantity of 500 and a discount of 3%” (Figure 2-3d)

```
SELECT L_PARTKEY
   FROM LINEITEM
WHERE
   L_SHIPDATE = ″23.12.1997″ AND
   L_DISCOUNT = 0.03 AND
   L_QUANTITY = 500
```

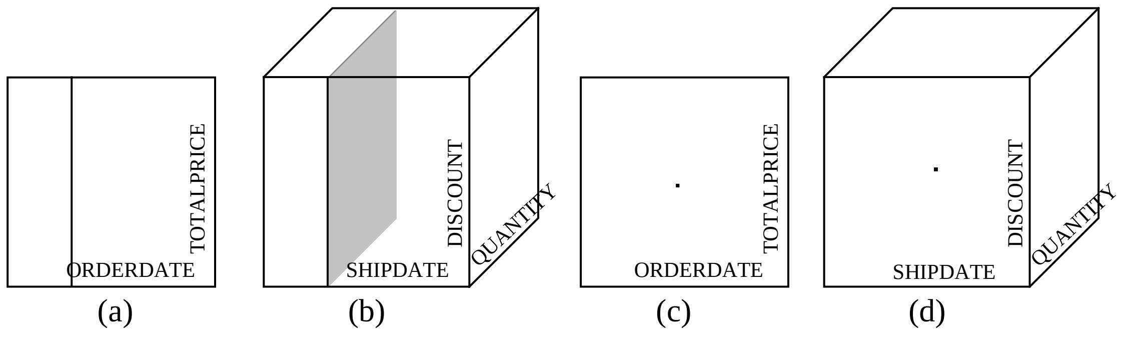


Figure 2-3: Partial Match Queries (a, b) and exact match queries (c, d)

2.2.2 Range Queries

A *range query* restricts all index candidates to an interval. For a pair of tuples y, $z \in \Omega$ we construct the query box $Q = [[y, z]]$. The result set RQ(Q, R) of a partial range query is:

$$\mathrm{RQ}(Q, R) = \{x \in R \mid (x_1,...,x_d) \in Q)\}.$$

Note that query boxes are multidimensional intervals and therefore iso-oriented. Moreover, the restriction in one dimension is independent of the restriction in all other dimensions.

Example 2-2a:

“return the number of orders in 1997 with a total price between 10000 and 20000” (Figure 2-4a)

```
SELECT COUNT(O_ORDERKEY)
   FROM ORDER
WHERE
   O_ORDERDATE >= ″1.1.1997″ AND
   O_ORDERDATE <= ″31.12.1997″ AND
   O_TOTALPRICE >= 10000 AND
   O_TOTALPRICE <= 20000
```

Example 2-2b:

“calculate the revenue for all shipments in 1997 with a quantity of less then 500 parts and a discount between 3% and 5%” (Figure 2-4b)

```
SELECT SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT))
   AS REVENUE
   FROM LINEITEM
WHERE
   L_SHIPDATE BETWEEN ″1.1.97″ AND ″31.12.97″
   AND L_DISCOUNT BETWEEN 0.03 AND 0.05
   AND L_QUANTITY < 500
```

A range query is called *partial range query*, if some attributes are not restricted, i.e. $y_i = -\infty$ and $z_i = +\infty$ for some $i \in D$. Since we are dealing with finite domains, it suffices to use the maximum boundaries instead of $+\infty$ and $-\infty$, i.e, $y_i = \lambda_i$ and $z_i = \upsilon_i$.

Example 2-2c:

“calculate the total price of all orders of 1997” (Figure 2-4c)

```
SELECT SUM(TOTALPRICE)
FROM ORDER
WHERE
   O_ORDERDATE >= ″1.1.1997″ AND
   O_ORDERDATE <= ″31.12.1997″
```

Example 2-2d:

“list all shipments in May ’97 with a discount between 3% and 5%” (Figure 2-4d)

```
SELECT L_ORDERKEY
FROM LINEITEM
WHERE
   L_SHIPDATE BETWEEN ″1.5.97″ AND ″31.5.97″ AND
   L_DISCOUNT BETWEEN 0.03 AND 0.05
```

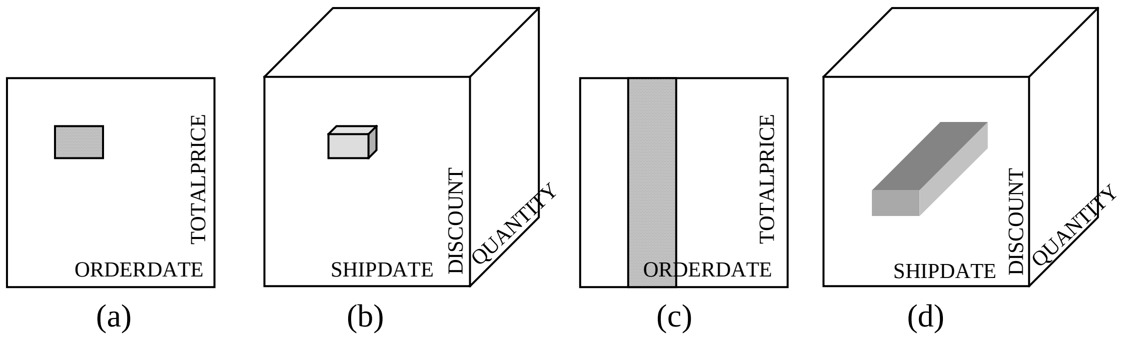


Figure 2-4: Range queries (a, b) and partial range queries (c, d)

Partial match queries and exact match queries as described in Section 2.2.1 are special cases of partial range queries. For partial match queries either $y_i = z_i$ or both $y_i = -\infty$ and $z_i = +\infty$ for all $i \in D$. For point queries $y_i = z_i$ holds for all $i \in D$. Thus partial range queries can be classified depending upon whether the restriction in each dimension is a point, an interval or whether this dimension is left unspecified. Table 2-1 shows this classification for the different types of partial range queries described in the previous sections, where d is the number of dimensions and m and n are integer numbers such that $m + n \leq d$.

	point	interval	unspecified
exact match query	d	-	-
partial match query	n	-	d-n
range query	-	d	-
partial range query	m	n	d-m-n

Table 2-1: Number of dimensions restricted to points, intervals or unrestricted for several types of partial range queries

2.2.3 Range Query Sets and Arbitrary Query Spaces

A range query set consists of a union of query boxes $S = Q_1 \cup ... \cup Q_n$. The result of the range query set S is the set of tuples

$$\mathrm{RQS}(S, R) = \{x \in R \mid (x_1,...,x_d) \in S)\}.$$

Range query sets can be used to model arbitrary query spaces, since every non iso-oriented query space can be decomposed into or approximated (covered) by a set of iso-oriented query boxes. Figure 2-5(c and d) shows such query volumes. For certain predicates, this set might become very large. In these cases it might be useful to construct a cover consisting of a fixed number of query boxes that include the query space.

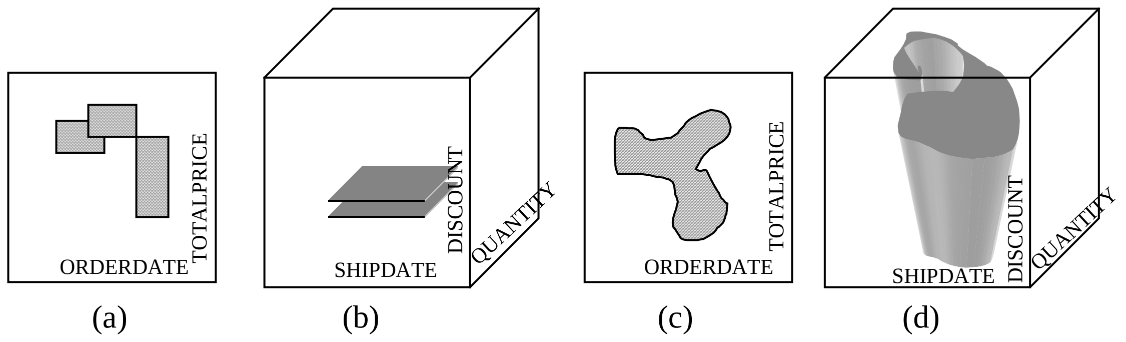


Figure 2-5: Sets of query boxes (a, b) and arbitrary query spaces (c, d)

Example 2-3a:	**Example 2-3b:**

"list all orders with a total price between 10000 and 20000 in 1997 together with all orders with a total price between 15000 and 25000 between Oct. 97 and Sept. 98 together with all orders between Oct. 98 and Mar. 99 with a total price between 5000 and 15000" (Figure 2-5a) (SQL omitted due to the length of the statement)	"list all parts that were shipped in May 1997 with a quantity between 1000 and 10000 pcs. per shipment and discounted with either 3% or 5%" (Figure 2-5b)
	`SELECT L_PARTKEY`
	`FROM LINEITEM`
	`WHERE`
	`   L_SHIPDATE BETWEEN ″1.5.97″ AND ″31.5.97″`
	`   AND L_DISCOUNT IN (0.03, 0.05)`
	`   AND L_QUANTITY BETWEEN 1000 AND 10000`

2.2.4 Nearest Neighbor Queries

A nearest neighbor query returns the tuples in the data base that have the least distance to a given point with respect to a specified distance function.[3] The distance function depends on the application. Typical distance functions for geometric nearest neighbor queries include the Euclidean distance function and the Manhattan chessboard distance. Another important application of nearest neighbor queries are preference queries, where personal preferences are specified by a point and one is interested to find the tuples in the database that best match our preferences.

Formally, a nearest neighbor query for a certain distance function can be described by

$$\text{NNQ}(x, R) = \{y \in R \mid \text{distance}(x, y) \text{ is minimal}\}$$

Nearest neighbor queries differ from the queries in the previous sections, since instead of a query box only one point is specified as input parameter. The space that needs to be retrieved from the database to answer such a query depends on the multi-dimensional distribution of the data. The result set of a nearest neighbor query often is a single point and in general is much smaller than that of range queries. However, in Section 4.4 we will show, that nearest neighbor queries may also be efficiently answered using a range query algorithm.

A one dimensional nearest neighbor query to find the nearest neighbor to value *b* for attribute *A* of relation *R* with the distance function < can be expressed in SQL by:

```
SELECT MIN(A) FROM
     SELECT MIN(A) FROM R WHERE A >= b
     UNION
     SELECT MAX(A) FROM R WHERE A <= b
```

[3] Note that the neighbor concept used in the context of nearest neighbor queries differs from the neighbor concept defined in Section 2.1.1: Whereas Section 2.1.1 defines a neighbor in space (*space neighbor*), here we mean a neighbor that is actually stored in the relation (*data neighbor)*. Only for dense relations, where each point of the base space is actually stored as tuple in the relation, the space neighbor and the data neighbor of a tuple are identical.

Example 2-4a:

"show the order that had a total price around 10000 and was shipped around 31.12.1998" (Figure 2-6a)

```
SELECT O_ORDERKEY
FROM ORDER
WHERE
O_ORDERDATE AROUND[4] ″31.12.1998″
AND
O_TOTALPRICE AROUND 10000
```

Example 2-4b:

"show the shipment that shipped a part with a discount around 4% and a quantity of around 1000 pieces on 23.12.97." (Figure 2-6b)

```
SELECT L_ORDERKEY
FROM LINEITEM
WHERE
L_SHIPDATE = ″23.12.97″ AND
L_DISCOUNT AROUND 0.04 AND
L_QUANTITY AROUND 1000
```

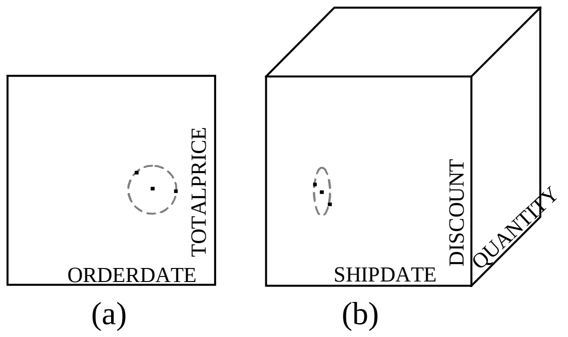


Figure 2-6: Nearest neighbor query

2.2.5 Further Queries

Especially in geometric applications, further queries are of interest, e.g., the intersection between two sets of objects or the union of two sets of objects. For two sets of extended objects, other examples are enclosure queries, containment queries and adjacency queries. [GG97] argue, that determining the intersection between two sets of objects provides an efficient filter step for answering any of these queries. Since the intersection between two sets of extended objects can efficiently be answered by range queries taking the extended objects as query volume, these problems can also be reduced to range queries.

Complex queries involve several tables, which are joined by some join condition. The most frequent method of joining tables is the equi-join, where matching values of attributes of several tables define the result of the join. Usually equi-joins are realized by sort-merge joins or hash-joins. In multidimensional space we can easily describe the processing of a sort-merge join: Each relation is processed in slices with respect to the join attribute. This allows to process the tuples of each relation in sort order of the join attribute. For identical join attributes in both tuples, the tuples are merged and the new tuple is added to the result set.

Processing a relation in sort order of any attribute is also necessary for further operations of the relational algebra such as sorting, projection and grouping and aggregation. We will discuss this processing method in more detail in Chapter 6.4.

[4] The keyword AROUND does not exist in SQL. We just use it here for convenience as a distance function for an attribute.

Example 2-5 (TPC-D Shipping Priority Query, Q3):

“This query retrieves the shipping priority and potential revenue of the orders having the largest revenue among those that had not been shipped as of a given date” (cf. [TPC97])

```
SELECT L_ORDERKEY, SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS REVENUE,
    O_ORDERDATE, O_SHIPPRIORITY
FROM CUSTOMER, ORDER, LINEITEM
WHERE
    C_MKTSEGMENT = ’FOOD’ AND
    C_CUSTKEY = O_CUSTKEY AND
    L_ORDERKEY = O_ORDERKEY AND
    O_ORDERDATE < ″1.5.98″ AND
    L_SHIPDATE > ″1.6.98″
GROUP BY L_ORDERKEY, O_ORDERDATE, O_SHIPPRIORITY
ORDER BY REVENUE DESC, O_ORDERDATE
```

For Example 2-5 we use C_CUSTKEY, C_MKTSEGMENT, L_ORDERKEY, L_SHIPDATE, O_ORDERKEY, O_ORDERDATE and O_SHIPDATE as index attributes. On each of the three tables an interval restriction in one attribute is imposed by the query. We first join CUSTOMER and ORDER via CUSTKEY by simultaneously processing the tuples of each relation in slices in the order indicated by the arrows in Figure 2-7. Afterwards we sort the intermediate result of CUSTOMER ⋈ ORDER on ORDERKEY and join it with LINEITEM. For each table we use the restriction to reduce the size of each slice.

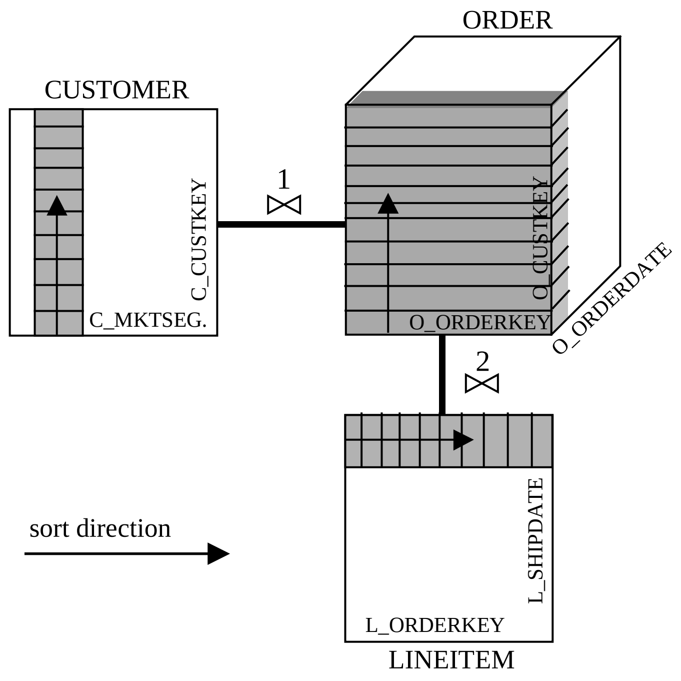


Figure 2-7: Joins and range queries in multidimensional space

2.2.6 Query Processing

An important task of query processing in RDBMS is to efficiently implement algorithms for the basic operations of the relational algebra. Usually these algorithms apply to particular storage structures or access methods. [Gra93] gives a concise survey of query processing. The selection operation is either implemented by a table scan or, if an index is available, by an index scan. Common access methods for indexes are B-Trees [BM72] and hash indexes. For retrieval intensive environments further indexing methods like bitmap indexes have been proposed (e.g., [OQ97], [Inf97]). If the selection condition specifies a range in a single attribute, a clustering index greatly speeds up query processing. Conjunctive selection conditions are efficiently processed by composite indexes, intersection of record pointers or multidimensional indexes. We will investigate processing of multi-attribute restrictions in more detail in Section 2.4 and Chapter 6.

The join operation is usually implemented by nested loop algorithms, join indexes, sort-merge algorithms, or hash algorithms. [ME92] surveys join processing in relational databases. Projection, union, intersection, and set difference are efficiently implemented by processing a relation in some sort order and then either use an index scan or merge-sort algorithm. Efficient sort operations and the use of restrictions to limit result sets are crucial to many query processing algorithms. Very often queries combine several operations of the relational algebra like join and restriction.

In Chapter 8 we will see how a multidimensional organization of a table will be used to efficiently process queries with multi-attribute restrictions and sort operations.

2.3 Access Methods

In this section we briefly describe the main characteristics of access methods that may be used to process the query types introduced in Section 2.2. Actual query processing strategies will then be described in Section 2.4.

Access methods transfer data from secondary storage in order to answer a query on a database. The most simple access method is a *full table scan* (FTS), which reads an entire relation and for each tuple checks the predicate φ of the query in main memory. *Indexes* are optional auxiliary data structures associated with a table. Given an index key value, the rows that contain that value can be directly located through the index. Generally, indexes are used to provide keyed access to rows within a table. The goal is to use the restrictions defined by a query to reduce the number of disk pages that have to be retrieved from secondary storage. A variety of indexes for efficient access to data stored in large databases has been implemented in commercial database systems or is being investigated by the research community. In commercial DBMS heap structures [Knu68, Knu73], hashing [FNP+79] and B-Trees ([BM72], [BU77], see [Com79] for a survey) are used to store tables. The most prevalent data structure is the B-Tree family, since it gives logarithmic performance guarantees with respect to the number of tuples stored in a table for the basic operations of insertion, deletion and point queries. In addition, B-Trees efficiently process one-dimensional range queries.

2.3.1 Characteristics of Secondary Storage

Secondary storage consists of a *stack* of rotating disks (cf. Figure 2-8). Each disk consists of a group of *tracks,* each of which consists of a number of blocks. A *block* is the smallest unit of transfer of the secondary storage. Each disk has a *read/write-head* (R/W-head) that radially moves over the disk. To read a page the R/W-head moves to the track containing that block. Since the disk is spinning, a block is read when it passes the R/W-head. Typical block sizes range from 512 Bytes to 8kB. DBMS usually define *database pages* each of which consists of a constant number of consecutive disk blocks. Thus database pages can be read with a single positioning of the R/W-head, i.e., by one random access. DBMS page sizes usually range from 2kB to 64kB. Since tuples are often smaller than a database page, a page in general holds more than one tuple.

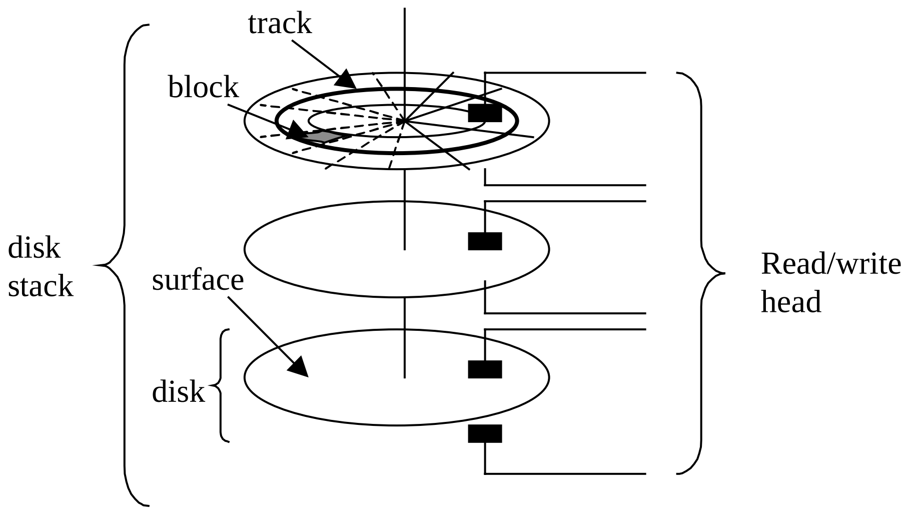


Figure 2-8: A hard disk

Each random access to a disk page takes some time t_π to position the R/W-head of the disk to the corresponding page, some time t_τ to transfer the page from disk to main memory and some time t_ξ to extract the relevant tuples from the page. t_π consists of the track positioning time t_χ, i.e., the time to position the R/W-head to the corresponding track, and the latency time t_λ, i.e., the time the disk needs to spin until the correct page appears under the R/W-head. t_π and t_τ are times spent for I/O, while t_ξ is time spent in the CPU. To distinguish between I/O-time and CPU-time we define $t_{\mathrm{I/O}} = t_\pi + t_\tau$ and $t_{\mathrm{CPU}} = t_\xi$. All in all we get the time to retrieve a page from secondary storage: $t_{\mathrm{PAGE}} = t_{\mathrm{I/O}} + t_{\mathrm{CPU}} = t_\pi + t_\tau + t_\xi$. For current hard disks and CPUs typical values are t_π = 10 ms, t_τ = 0,6ms and t_ξ = 0,4ms [IBM97]. So the positioning time of a hard disk and therefore the number of random page accesses is the limiting factor when retrieving a tuple.

2.3.2 Clustering

The idea of clustering is to store data that is likely to be used together, in physical proximity to reduce the number of I/Os necessary to retrieve the data. If the physical proximity is restricted to the tuples on a page, we speak of *tuple clustering*. If physical proximity in addition also holds between pages, we speak of *page clustering*.

In order to speed up joins, clustering might store the join partners of two relations physically close together. For range queries in attribute A_i, tuples should be stored in order of attribute A_i. Then one I/O is likely to retrieve a disk page storing several tuples that are necessary to create the result set of a query. Thus clustering reduces the number of random page accesses that are necessary to answer a query.

Definition 2-23 (tuple clustering; page clustering): *Tuple clustering* stores tuples of one or several relations on one disk 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 disk in insertion order. *Page clustering* in addition to tuple clustering also maintains physical clustering between disk pages.

The left part of Figure 2-9 shows unclustered data, tuple clustered data, and page clustered data. Page clustering is hardly feasible in OLTP environments, since each insertion or deletion requires a reorganization of the entire file. Therefore page clustering only exists statically, e.g. after mass loading process or after the reorganization of a relation.

Clustering needs a grouping function, i.e., an equivalence relation and possibly an ordering on the equivalence classes, to arrange the data on disk. In the following we only consider clustering the tuples of one relation. In this case clustering is useful for range restrictions. For single attribute range queries the clustering order is defined by the sort order on the restricted attribute. For multidimensional range queries there are several possibilities for orderings. Here we just give some orders as examples. The specific characteristics of these orders will be investigated in Chapter 3.

Example 2-6 (compound ordering, concatenated ordering):

We can create a multidimensional ordering of d attributes by concatenating the attributes in some order and using a lexicographic ordering between the attributes, whereas the ordering $<$ of each attribute is used to compare values of this attribute. This ordering is asymmetrical, since it favors the leftmost attributes of the concatenation. We call the ordering defined above *compound ordering* (or concatenated ordering). For $x_1 \in \{a,b,c,d\}$ and $x_2 \in \{a,b,c,d\}$ the 2-dimensional compound ordering $\lessdot_{A1^\circ A2}$ (see Section 3.1 for a definition of compound ordering) is:

aa $\lessdot_{A1^\circ A2}$ ab $\lessdot_{A1^\circ A2}$ ac $\lessdot_{A1^\circ A2}$ ad $\lessdot_{A1^\circ A2}$ ba $\lessdot_{A1^\circ A2}$ bb $\lessdot_{A1^\circ A2}$ bc $\lessdot_{A1^\circ A2}$ bd $\lessdot_{A1^\circ A2}$ ca $\lessdot_{A1^\circ A2}$ cb $\lessdot_{A1^\circ A2}$ cc $\lessdot_{A1^\circ A2}$ cd. $\lessdot_{A1^\circ A2}$ da $\lessdot_{A1^\circ A2}$ db $\lessdot_{A1^\circ A2}$ dc $\lessdot_{A1^\circ A2}$ dd.

Example 2-7 (Z-ordering):

Another way to order the two-dimensional tuples is Z-ordering $\prec$ (see Section 3.1 for a definition of Z-ordering):

aa $\prec$ ab $\prec$ ba $\prec$ bb $\prec$ ac $\prec$ ad $\prec$ bc $\prec$ bd $\prec$ ca $\prec$ cb $\prec$ da $\prec$ db $\prec$ cc $\prec$ cd $\prec$ dc $\prec$ dd.

Z-ordering will be described and analyzed in more detail in Chapter 3.

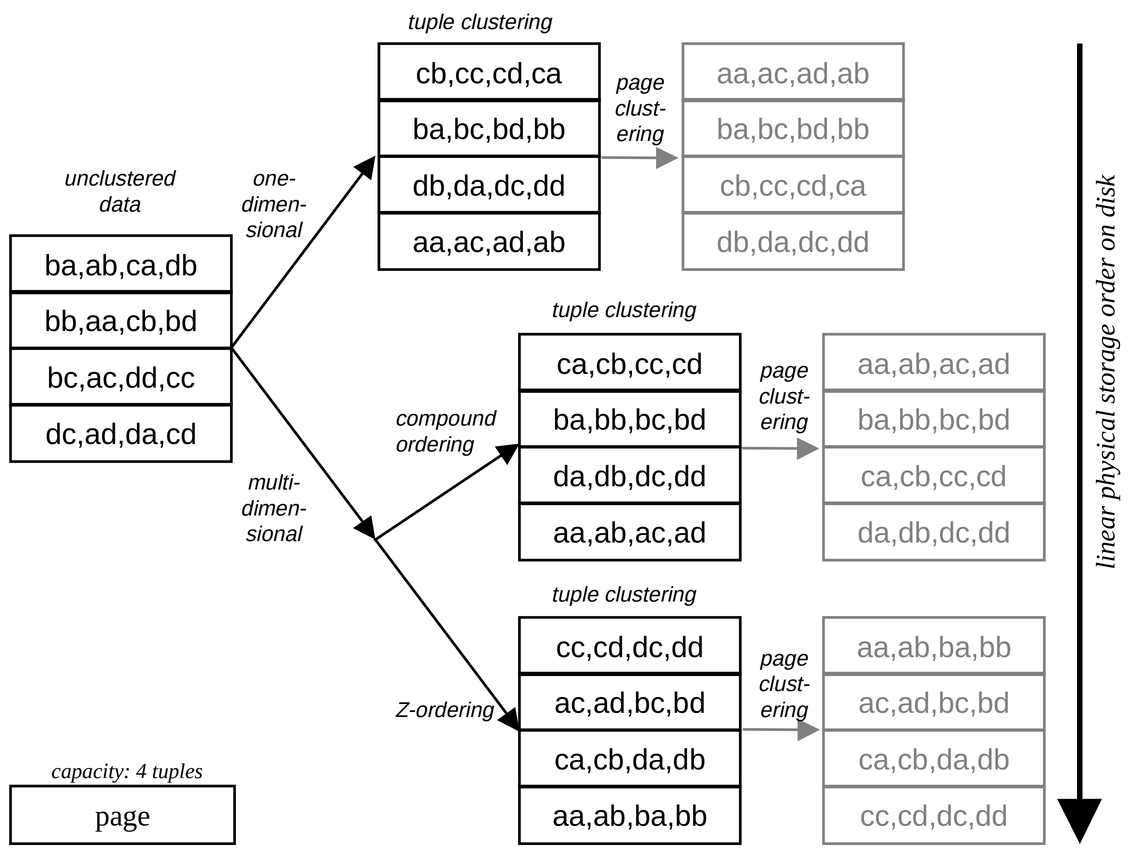


Figure 2-9: Clustering and dimensionality

Clustering and dimensionality are illustrated in Figure 2-9. Here 16 two-dimensional tuples {a,b,c,d} $\times$ {a,b,c,d} are stored on 4 disk pages. Each page is surrounded by a box in the figure, the data on each page is stored in the order as read from left to right. While the data are not ordered for unclustered data, they are at least ordered in the first attribute for one-dimensional clustering. For compound ordering and Z-ordering the data are clustered linearly with respect to these multidimensional orderings.

The importance of clustering can immediately be seen from the following retrieval cost estimations (we just consider one-dimensional clustering now. We will deal with the multi-dimensional case in the later chapters of this thesis). In accordance with [HR96] we use a cost model that takes I/O-time for random page accesses and I/O-time and CPU-time for page transfers into account. We assume that the prefetching strategy of the file system reads a physical cluster of L consecutive pages from disk with one random access into the read-ahead cache. This takes time $t_\pi + (t_\xi + t_\tau)\cdot L$. Reading k pages in consecutive order therefore takes:

$$c_{\text{scan}}(k) = \lceil k/L \rceil \cdot t_\pi + \max(k, L)\cdot (t_\tau + t_\xi)$$

2.3.2.1 Non-clustered Access / Random Access

Without clustering one random access to a page is necessary for each tuple. If we do not take caching of pages in main memory into account, each tuple requires a random access to the disk. For each random access it is necessary to position the R/W-head, to transfer the page and to extract the tuples from the page. Thus reading k pages with random access takes:

$$c_{random}(t_\pi, t_\tau, t_\xi, k) = k \cdot (t_\pi + t_\tau + t_\xi)$$

If main memory cache is used, the performance of random accesses may be significantly improved:

- One way to achieve this is to keep the data pages in cache after their retrieval. Then a page does not need to be fetched again from disk if a further tuple of this page also has to be retrieved. This requires a big portion of main memory whose size in worst case is equivalent to the number of tuples in the result set multiplied with the size of a page (if each tuple is stored on a separate page).
- Another strategy is to identify each tuple by its row identifier, which is the physical location of the tuple on the disk. First the row ids of all tuples in the result set are determined by the index access. Then these row ids are sorted, and the pages are retrieved from disk in this sorted sequence of row ids. This ensures that each page is only accessed once. This strategy requires no main memory for disk pages, but here the locations need to be cached and sorted.

Both strategies are only applicable if the result set of the corresponding query is small. Otherwise not enough main memory will be available and the system will start swapping. This will lead to a performance that is worse than the time needed for scanning the entire relation.

2.3.2.2 Tuple Clustered Access

If tuples are stored in a tuple clustered way, except for the first page and the last page all tuples of a page do contribute to a result set specified by a range in the clustering order. If C is the capacity of one page in tuples, tuple clustering reduces the number of random accesses by a factor $1/C$. Thus for k tuples at most $\lceil k/C+1 \rceil$ pages need to be randomly accessed and k tuples are extracted from these pages. Therefore the theoretical retrieval time for k pages by tuple clustered access is:

$$c_{tuple}(C, t_\pi, t_\tau, t_\xi, k) = \min(\lceil k/C+1 \rceil, k) \cdot (t_\pi + t_\xi + t_\tau)$$

Tuple clustering is used by B-Trees to speed up range queries. In this case a B-Tree is used to physically organize a relation on secondary storage. Because of this organization such a relation is called *index organized table* (IOT) in many commercial DBMS [Ora97, IBM97]. We consequently also use this term to denote a clustering B-Tree.

2.3.2.3 Page Clustered Access

If tuples are stored in a page clustered way, prefetching techniques can be used to further reduce the number of random accesses. Each random access retrieves not only one, but L consecutive pages and stores them in cache memory. The next L - 1 page accesses do not require any I/O, since the data is already available in main memory. Thus the number of random accesses gets reduced by a factor of L:

$$c_{page}(C, L, t_\pi, t_{\tau,}, t_\xi, k) = \min(\lceil \lceil k / C \rceil / L \rceil + 1, k) \cdot t_\pi + \max(\lceil k / C \rceil, L) \cdot (t_\tau + t_\xi)$$

Example 2-8:

We investigate the theoretical retrieval times for random access, tuple clustered access and page clustered access for k = 1, 5, 10, 50, 100, 1000 and 10000 tuples. We assume a secondary storage with an average positioning time t_π=10 ms and an average data transfer time t_τ=0.6 ms per page. We further assume that C = 50 tuples fit on one data page. In the case of page-clustering we further assume that L = 16 pages are pre-fetched with one random access. Figure 2-10 shows the tremendous advantage of both types of clustering over random access. An interesting fact is that page clustered access gets CPU-bound very quickly. If we assume that processing the tuples on a page (extraction of the tuples from the retrieved page and transfer to the address space of the user process) takes t_ξ = 0,4µs, the CPU component of the formula for page clustered access exceeds the I/O component when retrieving more than k = 544 tuples. However, random access and tuple clustering are generally I/O-bound.

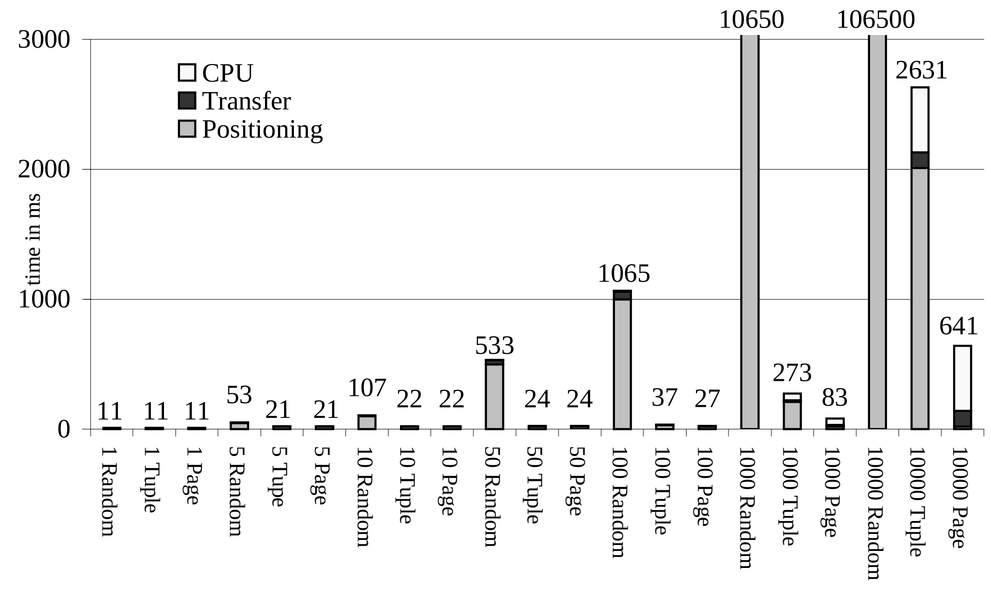


Figure 2-10: Theoretical performance of random access, tuple clustering and page clustering

Since a prefetching factor L basically means an increased page size, moving to larger page sizes may significantly speed up retrieval of large result sets. Therefore some database vendors offer database page sizes of up to 64kB. Exact match queries suffer from this strategy, however, since in this case a large amount of unnecessary data needs to be transferred to just retrieve a single tuple.

Although page clustering is hardly maintainable, it is important for one special case: A full table scan (FTS) without using any index is done in this way. Our performance measurements indicate that with a prefetching factor of $L = 16$ an FTS is about 10 times faster than an index scan of an entire relation [Pie98]. Thus, for queries with a selectivity of more than 10% an FTS is the best access method in a single user environment. With multiple transactions, it may not look so bad for indexes even in this case: An FTS puts an enormous load on the system, both in CPU time and I/O-time. In addition concurrent users may be prevented from updating tuples during an FTS.

2.3.3 Non-Clustering Indexes

Only one physical clustering order is possible for a table on secondary storage without introducing redundancy for storing a table several times. Secondary indexes are used for accessing tuples, when the restricted attributes are not included in the clustering order of the clustering primary index. Secondary indexes are a replica of a table that stores the *index attributes*, i.e., a certain subset of the attributes of a table, in some clustering order together with one additional attribute, which is a reference to the physical location of the entire tuple. We call this reference *row identifier* or *tuple identifier* (TID). Thus, when just restricting and retrieving index attributes, secondary indexes can also be regarded to be clustered. However, as soon as at least one non-index attribute needs to be retrieved, a random access to a page via the row identifier is necessary for each tuple.

Row identifiers of secondary indexes are often implemented as pointers to physical storage, i.e., a concatenation of a page number and an offset to a tuple on that page. In most cases a concatenation of integer numbers is sufficient to represent row ids. Oracle 8, for instance, uses row ids of 6 Bytes, which consist of three concatenated parts: data file, page number in file, and row number on page [Ora97].

So called “bitmap indexes” use a bitmap representation for the set of row identifiers having the same index key value [OQ97]. Thus, one bitmap is stored together with each different index key value and consists of as many bits as there are rows in the table. Each bit of the bitmap corresponds to one row and is set, if the corresponding row possesses that key value.

The physical location of a tuple stored in a table organized by a clustering B-Tree (IOT) may change because of page splits and page merges [BM72]. Therefore creating a secondary index on an IOT prevents using physical locations for tuple identifiers. Primary keys or some surrogate of them must be used instead. This requires one additional step of indirection (and often one additional page access), if a secondary index on an IOT is used to retrieve a tuple. Because of these complications some DBMS vendors do not allow to create a secondary index on an IOT [Ora97].

2.4 Answering Range Queries in present RDBMS

Most commercial RDBMS [Ora97, Inf98, IBM97, TAS98] do not use multidimensional indexes to process multi-attribute range queries. Thus we survey how traditional single-attribute access methods can be used to answer multidimensional range queries. For almost all commercial DBMS this means using B-Trees in combination with special query processing strategies.

2.4.1 Compound B-Trees

Many RDBMS vendors use a concatenation of multiple attributes to create an asymmetrical multidimensional clustering index with compound ordering (see Example 2-6). This results in extremely unbalanced query response times, when restricting different attributes (with restrictions of identical selectivities). The concatenation order of the attributes is a crucial factor for the query performance, since the first attribute in the concatenation order is preferred as the main clustering attribute. The asymmetry could only be cured by storing and maintaining at least d, for optimal performance factorial(d) index replica, each of which has a different attribute concatenation order. Because of its storage requirements this is hardly feasible for $d > 4$, since for 4 dimensions it is already necessary to store 24 replica of the index. So a query profile is necessary to decide which indexes to create. [GHR+97] analyzes this very difficult index selection problem for data warehousing scenarios. In addition to the tremendous storage requirements, it is not possible to use this approach in OLTP settings because of the manyfold response times for tuple insertion and deletion.

This type of index is called *concatenated index* or *compound index* by Informix, Oracle, DB2, TransBase or *star index* by RedBrick [Red97]. In the following we will use the term *compound B-Tree* for this index type. Very often this index is used as a non-clustering index, Oracle and TransBase may also use this type of index as an IOT to organize a relation.

2.4.2 Multiple Secondary B-Trees or Bitmap Indexes

Another approach is the so-called *inverted file* or *multiple secondary indexes* approach. Here a non-clustering secondary index is placed on each attribute. For answering a multidimensional range query $[[y, z]]$, the query is divided into d one-dimensional intervals $[y_1,z_1]$, ..., $[y_d,z_d]$, one for each attribute. Each of these intervals is then answered by the corresponding secondary index, resulting in a set of tuple identifiers (TIDs). The intersection of these d sets of TIDs determines the answer of the range query. For each TID in the intersection the corresponding tuple needs to be retrieved (see also Figure 2-11).

This approach has many performance problems: For d attributes, d indexes need to be maintained. This results in enormous storage requirements and increases OLTP response times for tuple insertion and deletion. Thus one usually selects a subset of these indexes, which then results in a difficult index selection problem. For OLAP databases this index selection problem (in combination with materialized view selection problem) is discussed in [GHR+97].

Moreover, intersecting tuple identifiers is a very expensive operation: If an attribute is not very selective, the set of TIDs for this attribute is very large. Each individual set must be stored and sorted. To make the query performance even worse: Since the data is not clustered, for small result sets an average of one random access to secondary storage is necessary to retrieve each tuple in the result set by its TID.

To improve the performance some database system vendors use bitmap indexes, since bitmaps provide a more compact representation of tuple identifiers for index attributes with low selectivity. Bitmaps avoid the sorting process, since each bitmap by definition is a list of tuple identifiers sorted in the order of the physical storage location. However, the necessity to access *d* bitmap indexes and the unclustered access to the tuples still remain as major performance bottlenecks. In addition, the index selection and maintenance problems still exist. When inserting a tuple, the length of each bitmap needs to be updated. This expensive insertion operation heavily limits the usability of bitmap indexes to applications with bulk updates.

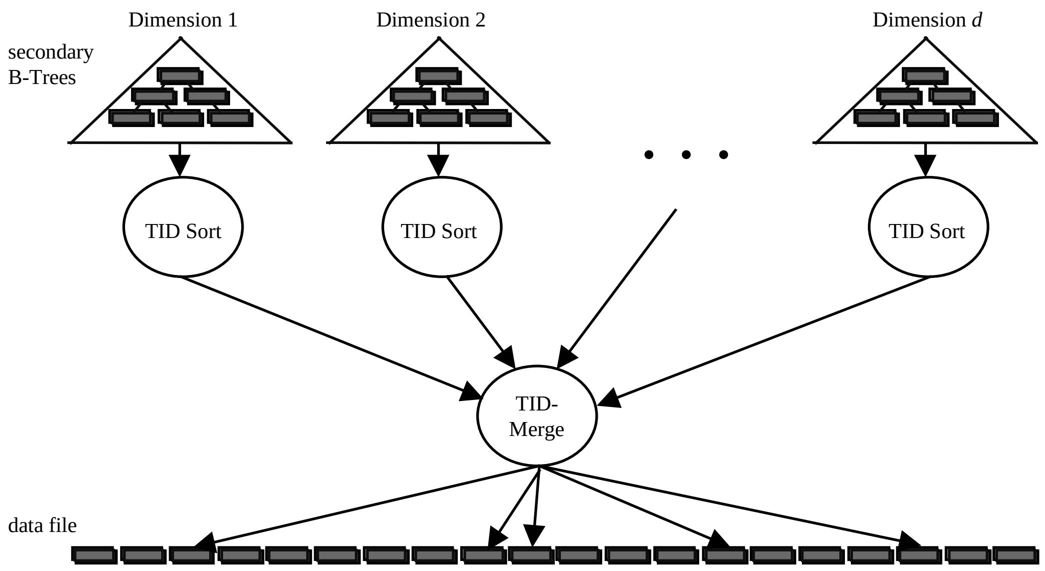


Figure 2-11: Using multiple B-Trees for answering range queries

2.4.3 Indexes for Processing Range Queries used in Present RDBMS

In this section we survey the access methods that, besides an FTS, commercial RDBMS use to answer multidimensional range queries.

Oracle implements both IOTs and non-clustering B*-Tree indexes and also offers a bitmap representation of row ids for non-clustering indexes. Bitmap indexes can also be used to index foreign columns, i.e., columns that are not part of the table, but are somehow joined to the table. As mentioned before, Oracle 8 does not allow one to create secondary indexes on tables organized as an IOT. TransBase offers IOTs and non-clustering B*-Trees. All indexes can either be used to index a single attribute or to combine several attributes in a compound B-Tree. DB2 only allows secondary B-Trees, which are in general not clustered. Clustering in DB2 can only be achieved statically by a reorganization tool. Although Oracle and Informix provide extenders for spatial data relying on kd-Trees [Ben75] or R-Trees [Gut84], these are not applicable for general indexing. Multidimensional access methods are not integrated in the core of any of these DBMS. Table 2-2 lists the standard index types of the DBMS Oracle, TransBase and DB2.

DBMS	**Oracle 8**			**TransBase 4.3**		**DB2 UDB**
index type	**secondary index**	**index organized table**	**bitmap index**	**primary index**	**secondary index**	**secondary index**
ordering	single, compound	single, compound	single	single, compound	single, compound	single, compound
clustering	no[5]	yes	no	yes	no[5]	no[5]
row id	6 Bytes concatenated integer number	-	bitmap	-	primary key or integer number	4 Bytes concatenated integer number
index concept	B*-Tree	B*-Tree	B*-Tree	B*-Tree	B*-Tree	B*-Tree

Table 2-2: Indexes in present RDBMS

[5] Clustering of the entire data tuple is only achieved by mass loading. This page clustering is destroyed when inserting tuples. For an optimal performance raw devices should be used, since the clustering of the DBMS might otherwise be destroyed by the file system. However, the index parts of the tuples are clustered in the index nodes of the B-Tree.

There is no excellent beauty that hath not some strangeness in the proportion.

(Francis Bacon)

Chapter 3

Multidimensional Space Partitioning

Space filling curves create a linearization of a multidimensional space and thus can be used for multidimensional space partitioning. Each point is indexed by its ordinal number on a space filling curve. Therefore we use space filling curves to address each point in multidimensional space. A special subspace (region) is constructed by intervals of these addresses. The region concept defines a totally ordered disjunctive partitioning of Ω, which may be used to define a clustering index for multidimensional data. In order to get a symmetrical multidimensional index, the multidimensional clustering of spatial neighbors should be preserved by the space filling curve. To preserve spatial proximity a space filling curve must be self similar, i.e., a fractal curve should be used. [Jag90] investigates several space filling curves for indexing. [Sag94] gives a concise mathematical treatment of space filling curves. In this chapter we investigate several space filling curves and some of their characteristics which are relevant to data processing. Since we rely on the Z-curve for our approach of multidimensional clustering and indexing, we investigate the Z-curve and its properties in more detail.

3.1 Space Filling Curves and Total Multidimensional Orderings

In the following we define two space filling curves, the compound curve (C-curve) and the Lebesgue curve (Z-curve). Without detailed analysis we also state some facts about the Hilbert curve (H-curve). We then investigate the characteristics continuity and monotonicity for one-dimensional orderings of a multidimensional space, as these properties are most relevant to data processing.

Definition 3-1 (space filling function): We call a function f : $S \subset \mathbb{N}_0 \to \Omega$ a space filling function, if f(S) = Ω is a bijective function.[6]

Lemma 3-1: A space filling function defines a one-dimensional ordering for a multidimensional space.

Proof:

The multidimensional space is ordered by f(0), f(1), f(2), ... □

Space filling curves are usually created by iterating some leitmotif to infinity [Sag90]. For practical applications in computer science it suffices to consider finite iterations. We define two curves for Ω with dimensions of identical cardinalities $r = r_1 = ... = r_d$ and $s = \log_2 r$ by the following formulas:

Definition 3-2 (C-value, C-address): For $x \in \Omega$ and the binary representation of each attribute $x_i = x_{i,s-1}x_{i,s-2}...x_{i,0}$ we define the compound lexicographic value (*C-value* or *C-address*) C(x):

$$\mathrm{C}(x) = \sum_{i=1}^{d} \sum_{j=0}^{s-1} x_{i,j} \cdot 2^{(i-1)\cdot s + j}$$

Definition 3-3 (Z-value, Z-address): For $x \in \Omega$ and the binary representation of each attribute $x_i = x_{i,s-1}x_{i,s-2}...x_{i,0}$ we define the *Z-value* (or *Z-address*) Z(x):

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

Without formal definition we call the values of the Hilbert-curve *H-values* or *H-addresses*.

[6] In mathematical treatments a space filling curve is the image of a *continuous* space filling function. We drop this requirement for our treatment of finite iterations of space filling functions for discrete finite spaces Ω.

Lemma 3-2: For $a \in \{0, ..., 2^{s \cdot d}-1\}$ with the binary representation $a = a_{s \cdot d-1}...a_0$ the inverse function $x = \mathrm{C}^{-1}(a)$ is calculated as:

$$x = (x_1, ..., x_d) \text{ with } x_i = \sum_{j=0}^{s-1} a_{i \cdot (d-1)+j} \cdot 2^j$$

Proof:

According to Definition 3-2: $a = a_{s \cdot d-1}...a_0 = \mathrm{C}(x) = \sum_{i=1}^{d} \sum_{j=0}^{s-1} x_{i,j} \cdot 2^{(i-1) \cdot s+j}$

Thus the dual numbers at the positions $a_{i \cdot (d-1)}$ ... $a_{i \cdot (d-1)+s-1}$ in that order form the dual numbers (i.e., binary string) representation of attribute x_i.

Therefore $x_i = \sum_{j=0}^{s-1} a_{i \cdot (d-1)+j} \cdot 2^j$

Applying the above formula for attributes x_1, ..., x_d builds the inverse $(x_1, ..., x_d) = \mathrm{C}^{-1}(a)$

□

Lemma 3-3: For $a \in \{0, ..., 2^{s \cdot d}-1\}$ with the binary representation $a = a_{s \cdot d-1}...a_0$ the inverse function $x = \mathrm{Z}^{-1}(a)$ is calculated as:

$$x = (x_1, ..., x_d) \text{ with } x_i = \sum_{j=0}^{s-1} a_{j \cdot d+i-1} \cdot 2^j$$

Proof:

According to Definition 3-3 $a = a_{s \cdot d-1}...a_0 = \mathrm{Z}(x) = \sum_{j=0}^{s-1} \sum_{i=1}^{d} x_{i,j} \cdot 2^{j \cdot d+i-1}$

Thus the dual numbers at the positions a_{i-1} a_{d+i-1} $a_{2 \cdot d+i-1}$ ... $a_{(s-1) \cdot d+i-1}$ in that order form the dual numbers (i.e., binary string) representation of attribute x_i.

Therefore $x_i = \sum_{j=0}^{s-1} a_{j \cdot d+i-1} \cdot 2^j$

Applying the above formula for attributes x_1, ..., x_d builds the inverse $(x_1, ..., x_d) = \mathrm{Z}^{-1}(a)$

□

Lemma 3-4: C(Ω) = Z(Ω) = {0, ..., $2^{s \cdot d}$-1} ⊂ $\mathbb{N}_0$

Proof:

(1) C(Ω) = {0, ..., $2^{s \cdot d}$-1}

From Definition 3-2 we derive that C(0,...,0) = 0. For $x \in \Omega \setminus (0,...,0)$ the C-value C(x) is larger than zero, since there is at least one attribute x_i whose binary representation has at least one bit set. Thus C(0,...,0) is the minimum of C(x).

With the same argument, the maximum value of C(x) is obtained for x = (r-1, ..., r-1) and C(r-1, ..., r-1) = $2^{d \cdot \log_2 r}$ = $2^{d \cdot s}$.

As a consequence of Lemma 3-2, C(x) is a bijective function.

Since $|\Omega| = r^d = 2^{d \cdot \log_2 r} = 2^{d \cdot s}$, the co-domain of C($x$) consists of $2^{d \cdot s}$ different values.

With the minimum and maximum values for C(x) we get: C(Ω) = {0, ..., $2^{s \cdot d}$-1}

(2) Z(Ω) = {0, ..., $2^{s \cdot d}$-1}

In analogy to (1) we derive Z(0,...,0) = 0 and Z(r-1, ..., r-1) = $2^{d \cdot s}$ from Definition 3-3. Z(x) is also bijective.

Thus we get Z(Ω) = {0, ..., $2^{s \cdot d}$-1}

□

Definition 3-4(compound curve and Z-curve): We call the image of C^{-1} *compound curve* (*C-curve*) and the image of Z^{-1} *Lebesgue curve (Z-curve)*.

Universes with different cardinalities for each dimension result in a more complex formula for C-values and Z-values. However, the basic properties of compound curves and Z-curves remain valid in this case as well. Without loss of generality we use a universe with identical cardinalities for all dimensions, because it is easier to understand the basic ideas and the formulas get less complex.

Figure 3-1 shows the ordering defined by three different space filling functions, the C-curve (a), the Z-curve (b), and the Hilbert curve (c).

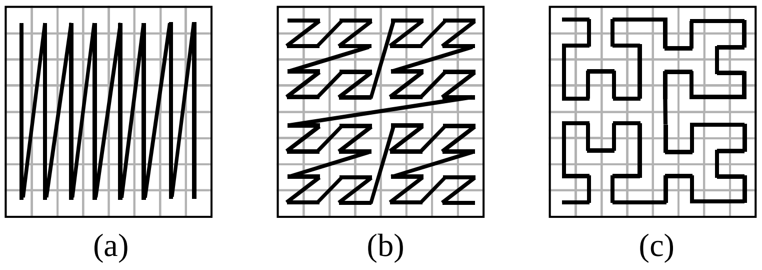


Figure 3-1: Space filling curves

Lemma 3-5: The C-curve creates the ordering $<_{Ad\circ...\circ A2\circ A1}$ on the multidimensional space Ω.

Proof:

According to Definition 3-2 C-values result in a binary representation of each attribute in the form

$$C(x) = x_{d,s-1}x_{d,s-2}...x_{d,0}x_{d-1,s-1}...x_{d-1,0}...x_{1,s-1}..x_{1,0}$$

For $x \in \Omega$ this is identical to the binary concatenation of the attributes in the order $x_d \circ ... \circ x_2 \circ x_1$. Thus, for $x, y \in \Omega$

$$C(x) < C(y) \Leftrightarrow x_d \circ ... \circ x_2 \circ x_1 < y_d \circ ... \circ y_2 \circ y_1 \Leftrightarrow x <_{xd\circ...\circ x2\circ x1} y$$

□

Definition 3-5 (Z-ordering, $\prec$): We call the ordering of the multidimensional space defined by Z-values *Z-ordering* and use $\prec$ to denote Z-ordering.

We call the lexicographic ordering $\prec$ on the steps of Z-addresses Z-ordering [OM84], since a path through ordinal numbers in each step reflects the letter “Z” in the two-dimensional case (see Figure 3-1b).

Lemma 3-6 (C-distance of two points): For two points $x, y \in \Omega$ with $x <_{A1\circ A2,..., \circ Ad} y$ their distance on the C-curve is:

$$\text{C-distance}(x, y) = \text{distance}(x, y, <_{A1\circ A2,..., \circ Ad}) = |C(y) - C(x)|$$

Proof:

The proof is a direct consequence of Definition 2-4 and Lemma 3-5.

□

Lemma 3-7 (Z-distance of two points): For two points $x, y \in \Omega$ with $x \prec y$ their distance on the Z-curve is:

$$\text{Z-distance}(x, y) = \text{distance}(x, y, \prec) = |Z(y) - Z(x)|$$

Proof:

The proof is a direct consequence of Definition 2-4 and Definition 3-5.

□

3.2 Properties of Space Filling Curves

After defining the terms continuity and monotonicity for space filling curves, we present proof that the compound curve and the Z-curve are not continuous but monotonous.

Definition 3-6 (continuity): A linear ordering $\leq$ of a multidimensional space $(\Omega, \trianglelefteq)$ is continuous, if, and only if:

for every $x, y \in \Omega$:

address(x) and address(y) are $\leq$-neighbors $\Rightarrow$ x and y are $\trianglelefteq$-neighbors

Lemma 3-8: The compound curve and the Z-curve are not continuous.

Proof:

(1) C-Curve

For $\Omega = \{0,..., r\text{-}1\} \times ... \times \{0,..., r\text{-}1\}$ the C-values r-1 and r are neighboring compound addresses. However, $C^{-1}(r\text{-}1) = (r\text{-}1, 0, ..., 0)$ and $C^{-1}(r) = (0, 1, 0, ..., 0)$ are not neighbored in Ω.

(2) Z-Curve

The Z-addresses 1 and 2 are neighboring Z-addresses. However, $Z^{-1}(1) = (1, 0, 0, ..., 0)$ and $Z^{-1}(2) = (0, 1, 0, ..., 0)$ are not neighbored in Ω.

□

Definition 3-7 (monotonicity): A one-dimensional $<$ ordering of a multidimensional space $(\Omega, \triangleleft)$ is monotonic, if and only if:

for every $x, y \in \Omega$: $x \triangleleft y \Rightarrow$ address(x) $<$ address(y)

Lemma 3-9: The compound curve and the Z-curve are monotonic.

Proof:

If $x \triangleleft y$, then for all dimensions $j \in D$ either $x_j = y_j$ or $x_j < y_j$. If $x_i < y_i$ for dimension i, then there exists a bit position k, so that for $k < a$ and $x_{i,k} < y_{i,k}$ and $x_{i,a} = y_{i,a}$. With Definition 3-2 this immediately yields $C(x) < C(y)$, with Definition 3-3 we obtain $Z(x) < Z(y)$.

□

3.3 Symmetry of Space Filling Curves

The Z-curve has the important property that in many cases the spatial proximity of points is preserved. However, sometimes the distance of two points in Z-values is shorter than the actual spatial distance (distance shrinking).[7] In other cases the Z-distance may be larger than the spatial distance (distance enlargement).

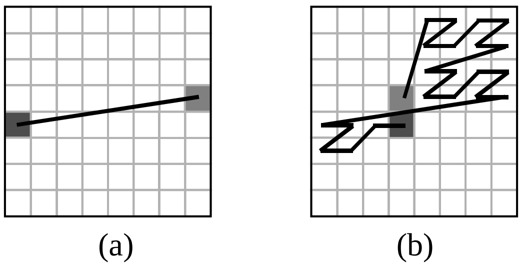


Figure 3-2: Distance shrinking and distance enlargement

Note that for the domains Ω_i, $i \in D$, the distance between two neighboring values is 1.

Definition 3-8 (successor and predecessor of a value): For Ω_i, $i \in D$, and $a \in \Omega_i$ we define the functions

$$\text{succ}(a) = a + 1, \text{ if } a < \max(\mathbb{D})$$

$$\text{pred}(a) = a - 1, \text{ if } a > \min(\mathbb{D})$$

Definition 3-9 (successor and predecessor of a tuple in one dimension): Since the domain of each attribute of a tuple is totally ordered, for $i \in D$ we can extend the definition of successors and predecessors to multidimensional tuples $x \in \Omega$:

$$\text{succ}_i(x)=(x_1,..., x_{i-1}, \text{succ}(x_i), x_{i+1},..., x_d)$$

$$\text{pred}_i(x)=(x_1,..., x_{i-1}, \text{pred}(x_i), x_{i+1},..., x_d)$$

Definition 3-10 (neighbors in one dimension of Ω): For attribute A_i, $i \in D$, of a tuple $x \in \Omega$ we define:

$$\text{neighbors}_i(x) = \begin{cases} \{\text{pred}_i(x)\} & , x_i = r_i - 1 \\ \{\text{succ}_i(x)\} & , x_i = 0 \\ \{\text{pred}_i(x)\} \cup \{\text{succ}_i(x)\} & , \text{otherwise} \end{cases}$$

Note that $\triangleleft - \text{neighbors}(x) = \bigcup_{i=1}^{d} \text{neighbors}_i(x)$.

[7] Note that distance shrinking does not occur for the Hilbert curve, since a neighbor on the Hilbert curve is always a neighbor in multidimensional space.

Definition 3-11 (average neighbor distance for a point in one dimension): For a point x we define *average neighbor distance* of a neighbor with respect to dimension *i* for the compound curve resp. Z-curve:[8]

$$\text{C-nd}_i(x) = \text{avg}(\{\text{C-distance}(x, y) \mid y \in \text{neighbors}_i(x)\})$$

resp.

$$\text{Z-nd}_i(x) = \text{avg}(\{\text{Z-distance}(x, y) \mid y \in \text{neighbors}_i(x)\})$$

Lemma 3-10: For a point *x* the average distance of a neighbor on the C-curve with respect to dimension *i* is:

$$\text{C-nd}_i(x) = r^{i-1}$$

Proof:

The proof is a direct consequence of Definition 3-2. □

For the following proofs we define a notion how to extract a bit from a value or an expression:

Definition 3-12 (bit of an expression): For any expression (*e*) yielding a scalar attribute we denote the *j*th rightmost bit of the result of (*e*) by $(e)_j$. For any expression (*e*) yielding a tuple we use $(e)_{i,j}$ to denote the *j*th rightmost bit of A_i of the result of (*e*) for any $i \in D$ and $j \in \{0, ..., s\text{-}1\}$.

Example 3-1:

$(2+3)_2 = (101_2)_2 = 1$

$(1+1)_0 = (10_2)_0 = 0$

Definition 3-13 (Δ_i): For multidimensional domains we define $\Delta_i = (x_1, ..., x_d)$ with $x_j = 0$ for all $j \in D\backslash\{i\}$ and $x_i = 1$.

Lemma 3-11: For a point *x* the average distance of a neighbor on the Z-curve with respect to dimension *i* is

$$\text{Z-nd}_i(x) = \begin{cases} 2^{i-1} & , \quad x_i = 0 \\ 2^{i-2} \cdot \sum_{j=0}^{s-1} \left((x_i+1)_j - (x_i-1)_j\right) \cdot 2^{j \cdot d} & , \quad x_i \in \{1, ..., r-2\} \\ 2^{i-1} & , \quad x_i = r-1 \end{cases}$$

[8] Note that according to Definition 3-10 each point has one or two neighbors per dimension, depending on whether it meets the border with respect to that dimension or not.

Proof:

A point $x = (x_1, ..., x_d)$ has at most two neighbors in dimension i, namely the two points $x - \Delta_i = (x_1,...,x_{i-1},x_i-1,x_{i+1},...,x_d)$ and $x + \Delta_i = (x_1,...,x_{i-1},x_i+1,x_{i+1},...,x_d)$

If $x_i = 0$ or $x_i = r$-1 there is only one neighbor, otherwise two neighbors exist.

For $x_i = 0$ only one neighbor exists. Thus we get:

$$\begin{aligned}
&\text{Z-nd}_i(x) = \\
&= Z(x+\Delta_i) - Z(x) = \\
&= \sum_{j=0}^{s-1}\sum_{i=1}^{d}(x+\Delta_i)_{i,j}\cdot 2^{j\cdot d+i-1} - \sum_{j=0}^{s-1}\sum_{i=1}^{d} x_{i,j}\cdot 2^{j\cdot d+i-1} = \\
&= \sum_{j=0}^{s-1}(x_i+1)_j\cdot 2^{j\cdot d+i-1} - \sum_{j=0}^{s-1} x_{i,j}\cdot 2^{j\cdot d+i-1} = \\
&= 2^{i-1}\cdot \underbrace{\left(\sum_{j=0}^{s-1}(x_i+1)_j\cdot 2^{j\cdot d} - \sum_{j=0}^{s-1} x_{i,j}\cdot 2^{j\cdot d}\right)}_{=1 \text{ for } x_i=0} = \\
&= 2^{i-1}
\end{aligned}$$

Analogously, for $x_i = r$ - 1 only one neighbor with respect to dimension i exists. Thus:

$$\begin{aligned}
&\text{Z-nd}_i(x) = \\
&= Z(x) - Z(x - \Delta_i) = \\
&= Z(x_1, ..., x_{i-1}, r\text{-}1, x_{i+1},..., x_d) - Z(x_1, ..., x_{i-1}, r\text{-}2, x_{i+1},..., x_d) = \\
&= 2^{i-1}
\end{aligned}$$

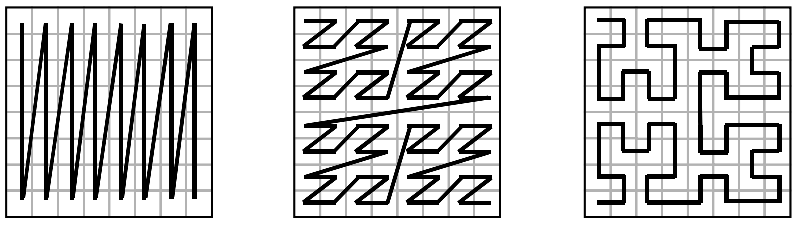
For $x_i \in \{1, ..., r\text{-}2\}$ two neighbors with respect to dimension i exist. We therefore get:

$$\begin{aligned}
&\text{Z-nd}_i(x) = \\
&= \frac{Z(x) - Z(x-\Delta_i) + Z(x+\Delta_i) - Z(x)}{2} = \\
&= \frac{Z(x+\Delta_i) - Z(x-\Delta_i)}{2} = \\
&= \frac{1}{2}\cdot\left(\sum_{j=0}^{s-1}\sum_{i=1}^{d}(x+\Delta_i)_{i,j}\cdot 2^{j\cdot d+i-1} - \sum_{j=0}^{s-1}\sum_{i=1}^{d}(x-\Delta_i)_{i,j}\cdot 2^{j\cdot d+i-1}\right) = \\
&= \frac{1}{2}\cdot\left(\sum_{j=0}^{s-1}(x_i+1)_j\cdot 2^{j\cdot d+i-1} - \sum_{j=0}^{s-1}(x_i-1)_j\cdot 2^{j\cdot d+i-1}\right) = \\
&= 2^{i-2}\cdot\left(\sum_{j=0}^{s-1}\left((x_i+1)_j - (x_i-1)_j\right)\cdot 2^{j\cdot d}\right)
\end{aligned}$$

□

Example 3-2:

Figure 3-3 (a and b) shows the C-values and Z-values of the space filling curves for the [0, 7]×[0, 7] universe of Figure 3-1. Although we have not derived a closed formula for the calculation of the values of the Hilbert curve, we call these values H-values and also include them in Figure 3-3c. Next to the right border and below the lower border of Figure 3-3 (a and b) we list the values of C-nd$_i$(x) and Z-nd$_i$(x). As the formulas suggest, Z-nd$_1$(x_1,x_2) is constant, when varying x_2 for a fixed x_1, Z-nd$_2$(x_1,x_2) is constant when varying x_1 for a fixed x_2 and Z-nd$_2$(x) = 2·Z-nd$_1$(x). C-nd$_1$(x) and C-nd$_2$(x) are constant over the entire universe. If one defined H-nd$_i$(x) in the same way as C-nd$_i$(x) and Z-nd$_i$(x), H-nd$_i$(x) is neither constant in the sense of Z-nd$_i$(x) nor C-nd$_i$(x).



ordinal numbers

	0	1	2	3	4	5	6	7
0	0	8	16	24	32	40	48	56
1	1	9	17	25	33	41	49	57
2	2	10	18	26	34	42	50	58
3	3	11	19	27	35	43	51	59
4	4	12	20	28	36	44	52	60
5	5	13	21	29	37	45	53	61
6	6	14	22	30	38	46	54	62
7	7	15	23	31	39	47	55	63

±1 ±1 ±1 ±1 ±1 ±1 ±1

±8 ±8 ±8 ±8 ±8 ±8 ±8

(a)

	0	1	2	3	4	5	6	7
0	0	1	4	5	16	17	20	21
1	2	3	6	7	18	19	22	23
2	8	9	12	13	24	25	28	29
3	10	11	13	14	26	27	30	31
4	32	33	36	37	48	49	52	53
5	34	35	38	39	50	51	54	55
6	40	41	44	45	56	57	60	61
7	42	43	46	47	58	59	62	63

±2 ±6 ±2 ±22 ±2 ±6 ±2

±1 ±3 ±1±11±1 ±3 ±1

(b)

	0	1	2	3	4	5	6	7
0	0	1	14	15	16	19	20	21
1	3	2	13	12	17	18	23	22
2	4	7	8	11	30	29	24	25
3	5	6	9	10	31	28	27	26
4	58	57	54	53	32	35	36	37
5	59	56	55	52	33	34	39	38
6	60	61	50	51	46	45	40	41
7	63	62	49	48	47	44	43	42

(c)

nd$_1$(x)

	0	1	2	3	4	5	6	7
0	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1
6	1	1	1	1	1	1	1	1
7	1	1	1	1	1	1	1	1

(d)

	0	1	2	3	4	5	6	7
0	1	2	2	6	6	2	2	1
1	1	2	2	6	6	2	2	1
2	1	2	2	6	6	2	2	1
3	1	2	2	6	6	2	2	1
4	1	2	2	6	6	2	2	1
5	1	2	2	6	6	2	2	1
6	1	2	2	6	6	2	2	1
7	1	2	2	6	6	2	2	1

(e)

	0	1	2	3	4	5	6	7
0	1	7	7	1	2	2	1	1
1	1	6	6	3	3	3	3	1
2	3	2	2	11	10	3	3	1
3	1	2	2	11	12	2	1	1
4	1	2	2	11	12	2	1	1
5	3	2	2	11	10	3	3	1
6	1	6	6	3	3	3	3	1
7	1	7	7	1	2	2	1	1

(f)

nd$_2$(x)

	0	1	2	3	4	5	6	7
0	8	8	8	8	8	8	8	8
1	8	8	8	8	8	8	8	8
2	8	8	8	8	8	8	8	8
3	8	8	8	8	8	8	8	8
4	8	8	8	8	8	8	8	8
5	8	8	8	8	8	8	8	8
6	8	8	8	8	8	8	8	8
7	8	8	8	8	8	8	8	8

(g)

	0	1	2	3	4	5	6	7
0	2	2	2	2	2	2	2	2
1	4	4	4	4	4	4	4	4
2	4	4	4	4	4	4	4	4
3	12	12	12	12	12	12	12	12
4	12	12	12	12	12	12	12	12
5	4	4	4	4	4	4	4	4
6	4	4	4	4	4	4	4	4
7	2	2	2	2	2	2	2	2

(h)

	0	1	2	3	4	5	6	7
0	3	1	1	3	1	1	3	1
1	2	3	3	2	7	6	2	2
2	1	3	3	1	7	6	2	2
3	27	26	23	22	1	4	6	6
4	27	26	23	22	1	4	6	6
5	1	3	3	1	7	6	2	2
6	2	3	3	2	7	6	2	2
7	3	1	1	3	1	1	3	1

(i)

Figure 3-3: C-values, Z-values and H-values

Definition 3-14 (cumulated neighbor distance for one dimension): The *cumulated neighbor distance* for dimension *i* is

$$\text{Z–nd}(i) = \sum_{x\in\Omega} \text{Z-nd}_i(x)$$

resp.

$$\text{C-nd}(i) = \sum_{x\in\Omega} \text{C-nd}_i(x)$$

Lemma 3-12: Z-nd(i) = $r^{d-1}\cdot 2^{i-1}\cdot\left(1+\frac{r^d-1}{2^d-1}\right)$

Proof:

For the proof we define $Z_1(a)=\sum_{j=0}^{s-1} a_j\cdot 2^{j\cdot d}$ for $a\in\mathbb{N}_0$

Then the average neighbor distance for the Z-curve is:

$$\begin{aligned}
&\text{Z-nd}(i)=\sum_{x\in\Omega}\text{Z-nd}_i(x)=\sum_{x_1=0}^{r-1}\sum_{x_2=0}^{r-1}\cdots\sum_{x_d=0}^{r-1}\text{Z-nd}_i(x_1,\ldots x_d)=\\
&=r^{d-1}\cdot 2^{i-1}\cdot\left(1+\sum_{x_i=1}^{r-2}\left(\frac{1}{2}\cdot\sum_{j=0}^{s-1}\left((x_i+1)_j-(x_i-1)_j\right)\cdot 2^{j\cdot d}\right)+1\right)=\\
&=r^{d-1}\cdot 2^{i-1}\cdot\left(1+\frac{1}{2}\cdot\sum_{x_i=1}^{r-2}\left(Z_1(x_i+1)-Z_1(x_i-1)\right)+1\right)=\\
&=r^{d-1}\cdot 2^{i-1}\cdot\left(1+\frac{1}{2}\cdot\left(Z_1(r-1)+Z_1(r-2)-Z_1(1)-Z_1(0)\right)+1\right)=\\
&=r^{d-1}\cdot 2^{i-1}\cdot\left(1+\frac{1}{2}\cdot\left(\sum_{j=0}^{s-1}(r-1)_j\cdot 2^{j\cdot d}+\sum_{j=0}^{s-1}(r-2)_j\cdot 2^{j\cdot d}-1-0\right)+1\right)=\\
&=r^{d-1}\cdot 2^{i-1}\cdot\left(1+\frac{1}{2}\cdot\left(\sum_{j=0}^{s-1}2^{j\cdot d}+\sum_{j=1}^{s-1}2^{j\cdot d}-1\right)+1\right)=\\
&=r^{d-1}\cdot 2^{i-1}\cdot\left(1+\frac{1}{2}\cdot\left(2\cdot\sum_{j=0}^{s-1}2^{j\cdot d}-2\right)+1\right)=\\
&=r^{d-1}\cdot 2^{i-1}\cdot\left(1+\sum_{j=0}^{s-1}2^{j\cdot d}\right)=r^{d-1}\cdot 2^{i-1}\cdot\left(1+\frac{2^{d\cdot s}-1}{2^d-1}\right)=r^{d-1}\cdot 2^{i-1}\cdot\left(1+\frac{r^d-1}{2^d-1}\right)
\end{aligned}$$

□

Lemma 3-13: C-nd(i) = $r^d \cdot r^{i-1}$

Proof:

$$\text{C-nd}(i) = \sum_{x \in \Omega} \text{C-nd}_i(x) = \sum_{x \in \Omega} r^{i-1} = |\Omega| \cdot r^{i-1} = r^d \cdot r^{i-1} \qquad \square$$

Definition 3-15 (degree of symmetry of a space filling curve): The degree of symmetry of a space filling curve is the negative standard deviation of the neighbor distances for each dimension, i.e.,

$$\text{C-symmetry}(\Omega) = \text{-std}(\{\text{C-nd}(i) \mid i \in D\})$$

resp.

$$\text{Z-symmetry}(\Omega) = \text{-std}(\{\text{Z-nd}(i) \mid i \in D\})$$

Example 3-3:

The compound curve for the universe of Figure 3-1 has a cumulated neighbor distance of C-nd(1) = 64 and C-nd(2) = 512. The Z-curve of Figure 3-1 has the following cumulated neighbor distances: Z-nd(1) = 176 and Z-nd(2) = 352. This results in a C-symmetry of –316,8 and a Z-symmetry of –124,5. Without further analysis we list the corresponding numbers for the Hilbert Curve: H-nd(1) = 230, H-nd(2) = 362, resulting in an H-symmetry of –93,3.

Theorem 3-1 (symmetry theorem): For practical values for $d \in \{2, ..., 10\}$ and $r \in \{2, ..., 10^9\}$ the Z-curve has a higher degree of symmetry than the compound curve.

Proof:

With

$$\Psi(d, r) = r^{d-1} \cdot \left(1 + \frac{r^d - 1}{2^d - 1}\right)$$

the cumulated neighbor distance for dimension i of the Z-curve is:

$$\text{Z-nd}(i) = 2^{i-1} \cdot \Psi(d, r)$$

We calculate the average μ_Z and the standard deviation σ_Z of the cumulated neighbor distance over all dimensions:

$$\mu_Z = \text{avg}(\{\text{Z}-\text{nd}(i) \mid i \in D\}) = \Psi(d,r) \cdot \frac{2^d - 1}{d}$$

$$\sigma_Z{}^2 = \text{std}(\{\text{Z}-\text{nd}(i) \mid i \in D\})^2 =$$

$$= \frac{1}{d} \cdot \sum_{i=1}^{d} \left(2^{i-1} \cdot \Psi(d,r) - \frac{\Psi(d,r) \cdot (2^d - 1)}{d} \right)^2 =$$

$$= \frac{\Psi^2(d,r)}{d} \cdot \left(\sum_{i=1}^{d} 4^{i-1} - 2 \cdot \frac{2^d - 1}{d} \cdot \sum_{i=1}^{d} 2^{i-1} + \sum_{i=1}^{d} \left(\frac{2^d - 1}{d} \right)^2 \right) =$$

$$= \frac{\Psi^2(d,r)}{d} \cdot \left(\frac{4^d - 1}{3} - 2 \cdot \frac{2^d - 1}{d} \cdot (2^d - 1) + d \cdot \left(\frac{2^d - 1}{d} \right)^2 \right) =$$

$$= \frac{\Psi^2(d,r)}{d} \cdot \left(\frac{4^d - 1}{3} - \frac{(2^d - 1)^2}{d} \right)$$

$$\sigma_Z = \Psi(d,r) \cdot \sqrt{\frac{1}{d} \cdot \left(\frac{4^d - 1}{3} - \frac{(2^d - 1)^2}{d} \right)} = r^{d-1} \cdot \left(1 + \frac{r^d - 1}{2^d - 1}\right) \cdot \sqrt{\frac{1}{d} \cdot \left(\frac{4^d - 1}{3} - \frac{(2^d - 1)^2}{d} \right)}$$

The cumulated neighbor distance for dimension *i* of the compound curve is:

$$\text{C-nd}(i) = \mathrm{r}^{i-1} \cdot r^d$$

We calculate the average μ_C and the standard deviation σ_C of the cumulated neighbor distance over all dimensions:

$$
\begin{aligned}
&\mu_C = \operatorname{avg}(\{C - nd(i) \mid i \in D\}) = \frac{r^d \cdot (r^d - 1)}{d \cdot (r-1)} \\
&\sigma_C{}^2 = \operatorname{std}(\{C - nd(i) \mid i \in D\})^2 \\
&= \frac{1}{d} \cdot \sum_{i=1}^{d} \left(r^d \cdot r^{i-1} - \frac{r^d \cdot (r^d - 1)}{d \cdot (r-1)} \right)^2 = \\
&= (r^d)^2 \cdot \frac{1}{d} \cdot \left(\sum_{i=1}^{d} (r^{i-1})^2 - 2 \cdot \frac{r^d - 1}{d \cdot (r-1)} \cdot \sum_{i=1}^{d} r^{i-1} + \sum_{i=1}^{d} \left(\frac{r^d - 1}{d \cdot (r-1)} \right)^2 \right) = \\
&= (r^d)^2 \cdot \frac{1}{d} \cdot \left(\frac{(r^2)^d - 1}{r^2 - 1} - 2 \cdot \frac{r^d - 1}{d \cdot (r-1)} \cdot \frac{r^d - 1}{r - 1} + \frac{(r^d - 1)^2}{d \cdot (r-1)^2} \right) = \\
&= (r^d)^2 \cdot \frac{1}{d} \cdot \left(\frac{(r^2)^d - 1}{r^2 - 1} - \frac{(r^d - 1)^2}{d \cdot (r-1)^2} \right) = \\
&= (r^d)^2 \cdot \frac{1}{d} \cdot \frac{r^d - 1}{r - 1} \cdot \left(\frac{r^d + 1}{r + 1} - \frac{r^d - 1}{d \cdot (r-1)} \right) \\
&\sigma_C = r^d \cdot \sqrt{\frac{1}{d} \cdot \frac{r^d - 1}{r - 1} \cdot \left(\frac{r^d + 1}{r + 1} - \frac{r^d - 1}{d \cdot (r-1)} \right)}
\end{aligned}
$$

For all practical values of $d \in \{2, ..., 10\}$ and $r \in \{2, ..., 10^9\}$ the ratio σ_Z/σ_C is less than 1. Due to its length, we omit the formal proof. The basic idea of the proof is, that σ_Z/σ_C shrinks monotonously, which is proven by building the derivations. We just show the graph of σ_Z/σ_C in Figure 3-4. The graph even suggests, that for $r \to \infty$ σ_Z/σ_C converges to a fixed value for each dimensionality *d*.

Overall, for $d \in \{2, ..., 10\}$ and $r \in \{2, ..., 10^9\}$ the degree of symmetry of the Z-curve is higher than the degree of symmetry of the compound curve. □

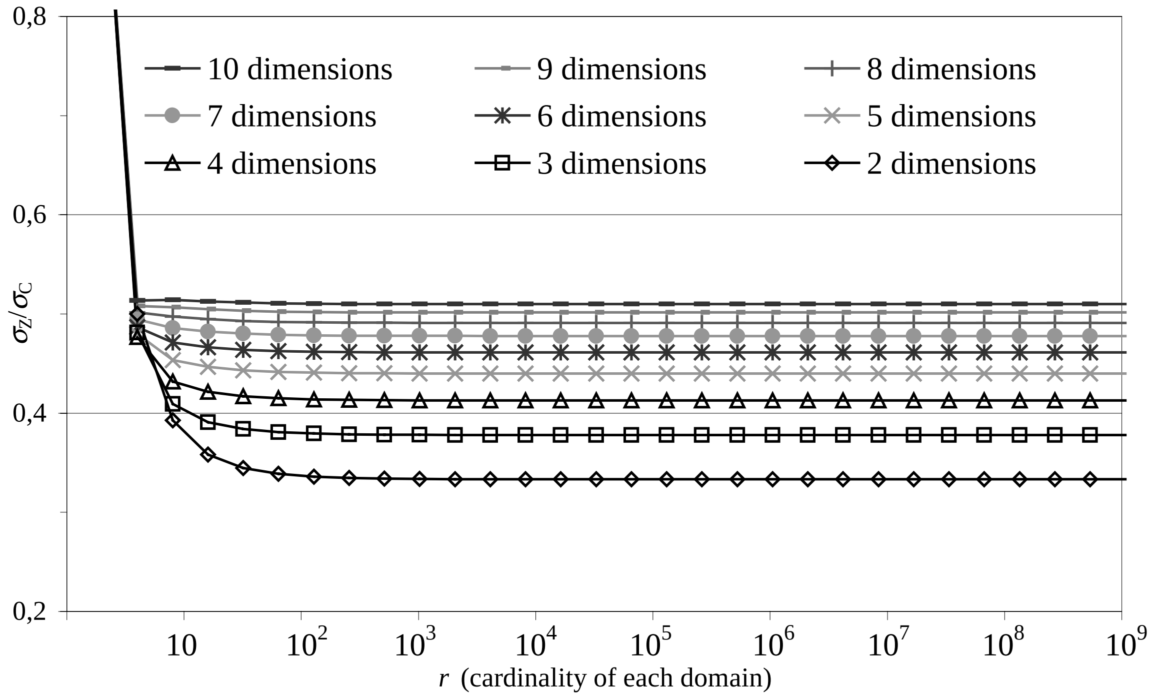


Figure 3-4: Relative degree of the symmetry σ_Z/σ_C

Hilbert curves are out of the scope of this thesis. Without proof we just state the following facts: The Hilbert curve is monotonous, continuous and has a higher degree of symmetry than the Z-curve. For our pilot implementation of the UB-Tree, we use the Z-curve, because of its easy implementation (see Section 5.3.1). Since the Hilbert curve exhibits better properties than the Z-curve, implementing the UB-Tree using a Hilbert curve would result in a better space partitioning and therefore in a better performance of the multidimensional index. This might be future work in this field. However, compared to compound clustering both the Hilbert curve and the Z-curve are far superior because of their higher degree of symmetry.

3.4 Regions covered by Space Filling Curves

In the following we define the region concept, a concept which defines a subspace of a multidimensional space which is covered by some part of a space filling curve.

Definition 3-16 (C-Region): A *C-region* $[\alpha :_C \beta]$ is the space covered by an interval of the C-curve and is defined by two C-addresses α and β.

Definition 3-17 (Z-Region): A *Z-region* $[\alpha :_Z \beta]$ is the space covered by an interval of the Z-curve and is defined by two Z-addresses α and β. We often write $[\alpha : \beta]$ instead of $[\alpha :_Z \beta]$, if it is clear from the context that we denote a Z-region.

For an 8×8 universe, i.e., s = 3 and d = 2, Figure 3-5a shows the C-region [7 $:_C$ 11] and Figure 3-5b shows the Z-region [4 $:_Z$ 20]. Figure 3-5c shows a C-region partitioning with 5 C-regions

$[0 :_C 6]$, $[7 :_C 11]$, $[12 :_C 31]$, $[32 :_C 51]$ and $[52 :_C 53]$. Figure 3-5d shows a partitioning with five Z-regions $[0 :_Z 3]$, $[4 :_Z 20]$, $[21 :_Z 35]$, $[36 :_Z 47]$ and $[48 :_Z 63]$. Figure 3-5e shows ten points which for page capacity of two points per page might be stored in a partitioned relation with the C-region partitioning of Figure 3-5c or the Z-region partitioning of Figure 3-5d.

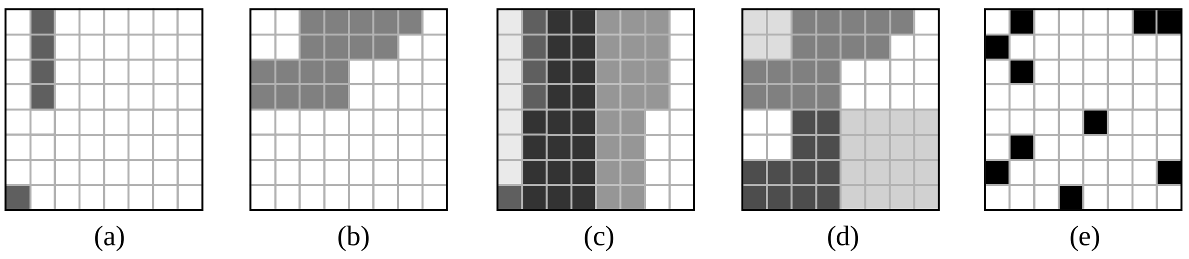


Figure 3-5: C-regions and Z-regions

Lemma 3-14: A Z-region $[\alpha : \beta]$ covers the multidimensional interval defined by the boundaries $Z^{-1}(\alpha)$ and $Z^{-1}(\beta)$ with $Z^{-1}(\alpha) \lhd Z^{-1}(\beta)$, i.e.,

$$\{Z(x) \mid x \in [[Z^{-1}(\alpha), Z^{-1}(\beta)]]\} \neq [\alpha : \beta]$$

but

$$\{Z(x) \mid x \in [[Z^{-1}(\alpha), Z^{-1}(\beta)]]\} \supset [\alpha : \beta]$$

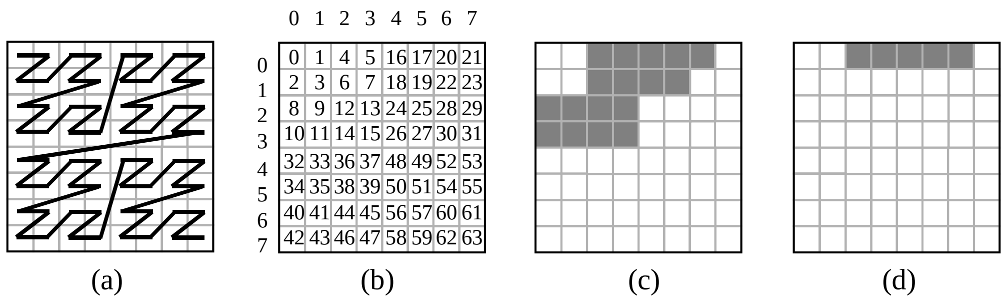
and thus

$$[[x, y]] \supset \{Z^{-1}(\alpha) \mid \alpha \in [Z(x) : Z(y)]\}$$

Proof:

(1) $\{Z(x) \mid x \in [[Z^{-1}(\alpha), Z^{-1}(\beta)]]\} \neq [\alpha : \beta]$

The Z-region $[Z(y) : Z(z)] = [4 : 20]$ (Figure 3-6c) defined by $y = (2,0)$ and $z = (6,0)$ is a superset of the space $[[y, z]] = [2,6] \times [0,0]$ (Figure 3-6d).



	0	1	2	3	4	5	6	7
0	0	1	4	5	16	17	20	21
1	2	3	6	7	18	19	22	23
2	8	9	12	13	24	25	28	29
3	10	11	14	15	26	27	30	31
4	32	33	36	37	48	49	52	53
5	34	35	38	39	50	51	54	55
6	40	41	44	45	56	57	60	61
7	42	43	46	47	58	59	62	63

Figure 3-6: Z-regions and query spaces

(2) $\{Z(x) \mid x \in [[Z^{-1}(\alpha), Z^{-1}(\beta)]]\} \supset [\alpha : \beta]$ and $[[x, y]] \supset \{Z^{-1}(\alpha) \mid \alpha \in [Z(x) : Z(y)]]$

This is a direct consequence of the monotonicity of the Z-curve in multidimensional space (Lemma 3-9). □

The above Lemma also holds for C-regions:

Lemma 3-15: A C-region $[\alpha :_C \beta]$ covers the multidimensional interval defined by the boundaries
$C^{-1}(\alpha)$ and $C^{-1}(\beta)$, i.e.,

$$\{C(x) \mid x \in [[C^{-1}(\alpha), C^{-1}(\beta)]]\} \neq [\alpha :_C \beta]$$

but

$$\{C(x) \mid x \in [[C^{-1}(\alpha), C^{-1}(\beta)]]\} \supset [\alpha :_C \beta]$$

and thus

$$[[x, y]] \supset \{C^{-1}(\alpha) \mid \alpha \in [C(x) :_C C(y)]\}$$

Proof:

In analogy to the proof of Lemma 3-14. □

3.5 Disconnected Z-Regions

Since the Z-curve is not continuous, two neighboring points on the Z-curve may not be neighboring points in the multidimensional space. This means that a Z-region can consist of spatially disconnected subsets of points.

Definition 3-18 (connection in space): We call two sets of points Q and P *spatially connected*, if there exists a pair of points $x \in Q$ and $y \in P$ that only differs in one attribute i and the distance between x_i and y_i is the limit of resolution, i.e., distance(x_i, y_i) = 1.

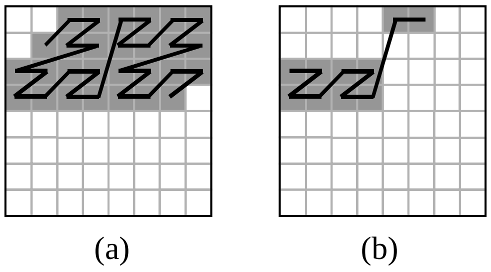


Figure 3-7: Connected and disconnected Z-regions

Theorem 3-2 (connection theorem): Any Z-region consists of at most two spatially disconnected sets of points and such Z-regions exist.

Proof:

For this proof we use the term point and address interchangeably. This is valid since there is a one-to-one mapping between addresses and points. We consider a Z-address to be a series of bits, whereby α_i denotes the i^{th} bit of address α read from left to right.

If a Z-region consists of disconnected sets of points, we can group the Z-addresses belonging to that Z-region into intervals each of which only contains a connected set of points. For *i* connected sets we consequently obtain *i* of these intervals.

We assume that a Z-region $[\alpha : \gamma^*]$ consists of three not connected sets of points. By introducing the addresses β and γ, we obtain three spatially disconnected intervals $[\alpha : \alpha^*]$, $[\beta : \beta^*]$, $[\gamma : \gamma^*]$ with $\beta = \alpha^* + 1$ and $\gamma = \beta^* + 1$. For Z_s-addresses with length *k* the interval bounds are uniquely represented by $\alpha^* = \beta_1 \ldots \beta_j 01 \ldots 1$, $\beta = \beta_1 \ldots \beta_j 10 \ldots 0$ and $\gamma = \gamma_1 \ldots \gamma_k$.

Now we distinguish two cases:

1. $\beta_1 \ldots \beta_j = \gamma_1 \ldots \gamma_j$: Since $\gamma \succ \beta$, there exists a position $a > j + 1$, so that $\gamma_a = 1$ and $\beta_a = 0$. By subtracting Δ from the attribute belonging to bit position *a* of the tuple cartesian(γ), we obtain a Z-address smaller than γ. However, by the subtraction we never reset a bit in a position greater than *a*. Since $a > j$, the resulting Z-address is always less then γ, but larger than β. Thus we always obtain Z-addresses in the interval $[\beta, \beta^*]$. This in contradiction to the assumption means that $[\beta : \beta^*]$ and $[\gamma : \gamma^*]$ are spatially connected.

2. $\beta_1 \ldots \beta_j \neq \gamma_1 \ldots \gamma_j$: This means that $\beta^* \succcurlyeq \beta_1 \ldots \beta_j 1 \ldots 1$. So every Z-address with the prefix $\beta_1 \ldots \beta_j 1$ is contained in $[\beta : \beta^*]$. Adding Δ to the attribute belonging to bit position j+1 of the tuple cartesian(α^*) sets bit j+1 of the Z-address of that tuple (and possibly resets some of the bits with a position greater than $j + 1$). This results in a Z-address greater than or equal to β, which has the prefix $\beta_1 \ldots \beta_j 1$. This Z-address is contained in the interval $[\beta, \beta^*]$. This in contradiction to the assumption means that $[\alpha : \alpha^*]$ and $[\beta : \beta^*]$ are spatially connected.

Summing up, the assumption of three disconnected sets of points in a Z-region is false for both cases. Combining this with Figure 3-7 proves the lemma. □

Spatial connection is an important property of Z-regions, since it guarantees spatial proximity independently of the dimensionality. This allows a Z-region partitioning to organize the multi-dimensional space while preserving multidimensional neighborship of data even for skewed data distributions. It enables to construct efficient algorithms for range queries and sorted reading.

3.6 Geometric View of Z-Region Partitioning

In Section 3.1 we algorithmically defined Z-region partitioning. Z-region partitioning can also be defined geometrically by the concepts of Z-areas and Z-regions [Bay96]. In this section we present this different view on Z-regions and prove the equivalence to the Z-region partitioning defined in Section 3.1. We therefore introduce a second notation of Z-addresses complementing the standard notation (Z_s-address, Z-address) as defined in Section 3.1. For the geometric view we define the concept of increment Z-addresses (Z_i-address). In addition we introduce the notion of step and length of a Z-address as well as the terms volume of Z-areas and Z-regions.

Definition 3-19 (Z-area; increment Z-address (Z_i-address)): We iteratively define a Z-area Λ as a special subspace of a *d*-dimensional cube as follows: Split the cube with respect to every dimension in the middle, resulting in 2^d subcubes numbered from 1 to 2^d. A Z-area Λ_1 of level 1 is the union of the first α_1 closed subcubes. α_1 determines Λ_1 uniquely. We call α_1 the Z-address of Λ_1 and write Λ_1 = area(α_1). The empty Z-area has the address ε_i. area(ε_i) = $\varnothing$. To enlarge a Z-area, we iteratively add a Z-area with Z-address $\alpha_2 \in \{0,1,...,2^d-1\}$ of the next subcube with number α_1+1. The Z-address of this enlarged Z-area Λ_2 is $\alpha_1.\alpha_2$, which is lexicographically larger then the address α_1 of area Λ_1. Next we may enlarge Λ_2 by adding an Z-area of the next subcube α_2+1 of α_2, etc.

We call the Z-address representation defined by this schema *increment representation* and append a subscripted “i” to address literals to denote that the literals are in increment representation. We write Z_i-address to denote a Z-address in increment representation.[9] In Section 3.6.1 we will define a standard representation for Z-addresses. If a distinction of standard representation and increment representation is not necessary in our explanations, we will use the term Z-address. Otherwise we will use Z_i-address or Z_s-address to denote a Z-address in increment representation respectively standard representation.

Example 3-4:

The left part of Figure 3-8 shows four Z-areas area($0.0.1_i$), area($1.3.2_i$), area(2.1_i), and area(3_i) of a two-dimensional universe. The shaded subcubes of the two-dimensional universe belong to the corresponding Z-area. area($0.0.1_i$) consists of 0 subcubes of the first level, 0 subcubes of the second level and 1 subcube of the third level. area($1.3.2_i$) consists of 1 subcube of the first subdivision level, 3 subcubes of the second subdivision level and 2 subcubes of the third subdivision level. In the same way area(2.1_i) consists of 2 subcubes of the first subdivision level and 1 subcube of the second subdivision level. area(3_i) merely consists of 3 subcubes of the first subdivision level.

[9] Actually the Z-addresses must be constructed depending on the multidimensional domain. That is, the Z-address representation depends on the data type (rational, irrational, complex, etc.). However, in practical computer systems irrational domains do not exist, but any data type can be mapped to a finite set of natural numbers. Thus it suffices to consider Z-addresses with integer numbers for each level.

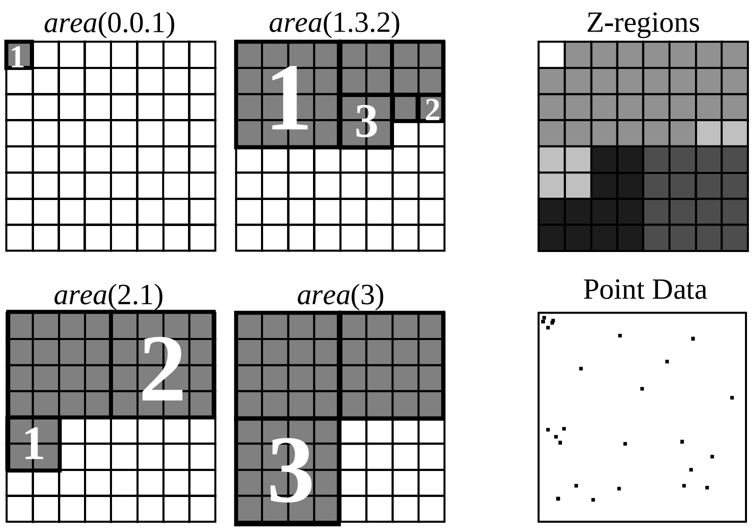


Figure 3-8: Z-areas and Z-regions

In the following we suppress trailing zeros of Z_i-addresses and denote Z-addresses by α, β, γ

Definition 3-20 (step and length of a Z_i-address): We call α_j the j^{th} *step* of the Z_i-address $\alpha = \alpha_1.\alpha_2. \ldots . \alpha_k$. We call k the *length* of Z_i-address α. By $\alpha_{j,i}$ we mean the i^{th} bit of the j^{th} step of α in binary representation.

Note that the volume of a subcube decreases exponentially with its step number. We therefore obtain a fine partitioning of the multidimensional space with relatively short Z_i-addresses.

For the following Lemma we loosen the definition of Z_i-addresses and allow the value of 2^d to occur in the last step of an Z_i-address. In this case some Z-areas have two Z_i-addresses.

Lemma 3-16: If for any $i > 0$ $\alpha_i \neq 2^d$, the Z_i-addresses $\alpha = \alpha_1. \ldots \alpha_i{+}1$ and $\alpha^* = \alpha_1. \ldots .\alpha_i.2^d$ define the same area.

Proof:

The union of the 2^d subcubes at subdivision level i+1 equals the α_i+1th subcube at level i. Since the previous steps of α and α^* are identical and thus describe the same area, the overall area is identical. □

Lemma 3-16 means that we can add steps to each Z_i-address until we reach the limit of resolution without changing the Z-area defined by this Z_i-address. The Z_i-addresses 2.1_i and 3_i in Figure 3-8 are merely a short form of the addresses $2.0.4_i$. and $2.3.4_i$ respectively. Removing trailing 2^d subcube numbers can be thought of like an overflow arithmetic in addresses taking place upon the completion of an entire subcube of the previous subdivision level when enlarging an area. Note that it is not possible to add steps to ε_i. We will use this relaxed Z_i-address definition as an auxiliary construct in some proofs of this chapter.

Lemma 3-17: The lexicographic ordering of Z-addresses (denoted by $\prec^*$) and set containment of areas in space (denoted by $\subseteq$) are isomorphic:

$$\text{area}(\alpha) \subseteq \text{area}(\beta) \Leftrightarrow \alpha \preccurlyeq^* \beta$$

Proof:

If and only if $\alpha = \beta$, then the same subcubes are added to area(α) and area(β) at each step. This results in area(α) = area(β).

(1) area(α)$\subset$ area(β) $\Leftarrow \alpha \prec^* \beta$

If $\alpha \prec^* \beta$, then there exists an index i so that $\alpha_j = \beta_j$ for all $j < i$ and $\alpha_i < \beta_i$. At subdivision point i we therefore add more subcubes to area($\beta_1. ... . \beta_{i\text{-}1}$) than to area($\alpha_1. ... . \alpha_{i\text{-}1}$). All subcubes that are added to α at later subdivision points are a subset of the subcube defined by β_i. Overall we get area(α) $\subset$ area(β).

(2) area(α)$\subset$ area(β) $\Rightarrow \alpha \prec^* \beta$

If area(α) $\subset$ area(β) there is a subdivision level j such that area($\alpha_1. ... .\alpha_{j\text{-}1}$) = area($\beta_1. ... . \beta_{j\text{-}1}$) and area($\alpha_1. ... .\alpha_j$) $\subset$ area($\beta_1. ... . \beta_j$). Thus a subdivision level j area($\beta_1. ... . \beta_j$) must consist of more subcubes than area($\alpha_1. ... .\alpha_j$). Because of the subcube numbering β_j then must be larger than α_j. Overall we get $\alpha \prec^* \beta$. □

Definition 3-21 (tuple, Z-address of a tuple): A *tuple* (or pixel) is a smallest possible subcube at the limit of the resolution, but the resolution may be chosen as fine as desired. The Z-address of a tuple is identical to the Z-address of the area defined by including the tuple as the last and smallest subcube contained in this area.

Theorem 3-3 (mapping between tuples and addresses): There exists a one-to-one map between Cartesian coordinates of a tuple and Z-addresses.

Proof:

The one-to-one mapping between Cartesian coordinates $(x_1,x_2, ..., x_d)$ of a d-dimensional tuple and its Z-address α is defined by the addressing scheme of Definition 3-19. As stated in Definition 3-21, a tuple is identified by the area containing the tuple as the last point. The mapping from tuples to Z-addresses can directly be derived from Definition 3-19.

Similarly, the area corresponding to α and its last point x are calculated by building the union of subcubes for Z-address α according to Definition 3-19. These two algorithms define the functions of the one-to-one map between Z-addresses and tuples. □

Definition 3-22 (mapping between tuples and addresses): We use the following notations for the mapping between a d-dimensional tuple $x = (x_1, \ldots, x_d)$ and its Z-address α:

$$\text{Z-address}(x) = \alpha \text{ and } \text{cartesian}(\alpha) = x$$

Lemma 3-18: Since the two maps are inverses of each other we get:

$$\text{cartesian}(\text{Z-address}(x)) = x \quad \text{and} \quad \text{Z-address}(\text{cartesian}(\alpha)) = \alpha$$

Definition 3-23 (Z-region): A Z-*region* is the difference between two Z-areas: If $\alpha \prec^* \beta$ then we define the Z-region between α and β as: $]\alpha : \beta] := \text{area}(\beta) \setminus \text{area}(\alpha)$, where "\" means "set difference".

Note that Z-regions as defined by Definition 3-23 are open below and closed above. Thus a set of ordered Z_i-addresses $\{\alpha_1, \ldots, \alpha_n\}$ builds a set of Z-regions $\{]\alpha_1 : \alpha_2], \ldots,]\alpha_{n-1} : \alpha_n]\}$. These Z-regions are disjoint and therefore partition – or tile – the universe.

Example 3-5:

The areas in Figure 3-8 are used to create five Z-regions: $]\varepsilon_i : 0.0.1_i]$, $]0.0.1_i : 1.3.2_i]$, $]1.3.2_i : 2.1_i]$, $]2.1_i : 3_i]$, $]3_i : 4_i]$. Each Z-region is shaded with a different gray.

Note that the lower bound address α of a Z-region $]\alpha : \beta]$ represents a point cart(α) that does not belong to $]\alpha : \beta]$, i.e., $\alpha \notin]\alpha : \beta]$ and $\beta \in]\alpha : \beta]$. Thus a Z-region can be regarded to be a one-dimensional interval of Z-addresses with respect to the Z-ordering $\prec^*$:

$$]\alpha : \beta] =]\alpha, \beta]$$

Thus a Z-region represents the space corresponding to all points located in the Z-interval $]\alpha, \beta]$.

3.6.1 Address Representation

The subcube enumeration defined in the previous section creates variable length Z_i-addresses, where the length of the address denotes the number of subdivisions. This address representation is useful when storing Z-addresses, since fewer subdivisions result in shorter addresses and thus in a better memory utilization. For address calculation and theoretical considerations, however, a different Z-address representation, the so-called standard representation of Z-addresses, is desirable:

To calculate the Z_s-address for a tuple x the address calculation has to proceed as follows: For each dimension $i \in D$ and each subdivision step $s \in \{1, \ldots, \log_2 r_i\}$ the current splitpoint $r_i/2^s$ must be compared with x_i. The comparison is a binary decision between $x_i < r_i/2^s$ and $x_i \geq r_i/2^s$. It is sufficient to use one bit to store each of these decisions in the address. If $x_i \geq r_i/2^s$, $r_i/2^s$ must be subtracted from x_i to correctly process the next step of the subdivision. The subdivision process continues up to the limit of resolution in each attribute. A Z_s-address then consists of a sequence of subdivision steps, each of which consists of one bit for each dimension.

It is easy to incorporate varying attribute lengths, i.e., attributes with different resolutions, into this algorithm: When the limit of the resolution in one attribute is reached at a certain subdivision step, this attribute is not used for the subdivision any further. The number of bits in each further step is reduced in this case.

Definition 3-24 (number of steps for an attribute): The number of steps for attribute A_i of a domain with cardinality[10] r_i is determined by its resolution:

$$\text{steps}(i) = \log_2 r_i$$

Definition 3-25 (length of a step): The number of dimensions in step k (i.e., the length of step k in bits) is:

$$\text{steplength}(k) = |\{\text{steps}(i) \mid \text{steps}(i) \geq k \text{ and } i \in \text{D}\}|$$

Definition 3-26 (standard representation of a Z-address (Z_s-address)): We call the representation of Z-addresses obtained by Algorithm 3-1 standard representation and append a subscripted “s” to address literals to denote that the standard representation is used. We write Z_s-addresss to denote addresses in standard representation. Each pass through the outer loop of the algorithm defines a step of a Z_s-address.

```
Input:  $x = (x_1, ..., x_d)$: $d$-dimensional tuple
        $r = (r_1, ..., r_d)$: cardinality vector for the
                              domain of the tuple
Output: $\alpha$: Z-address($x$) in standard representation

// the algorithm requires the dimensions to be
// sorted according to their resolution
// in descending order
for $s$ = 1 to steps(1)
   for $i$ = steplength($s$) to 1
      if $x_i < r_i/2^s$ then
         $\alpha_{s,d-i} = 0$
      else
         $\alpha_{s,d-i} = 1$
         $x_i = x_i - r_i/2^s$
      end if
   end for
end for
```

Algorithm 3-1: Z_s-address calculation by subdivision

[10] Note that we defined r_i to be 2^v for some $v \in \mathbb{N}_0$

Note that steps of a Z-address are numbered beginning with 1 from left to right in this thesis, whereas the bits of a step are numbered beginning with 0 from right to left. So the leftmost bit of a step corresponds to the rightmost dimension in the order of dimensions.

Z_s-addresses always describe an area with positive volume, i.e., at least the point at the limit of the resolution must be included in the Z-area. Thus the empty Z-area $\emptyset$ does not have a Z_s-address, i.e., there is no counterpart to ε_i.

Example 3-6:

For a 3d-universe with $r_1 = 4$, $r_2 = 4$, $r_3 = 8$ we obtain steps(1) = 2, steps(2) = 2 and steps(3) = 3. The length of the steps are: steplength(1) = 3, steplength(2) = 3 and steplength(3) = 1. Calculating the address for the tuple $(x_1, x_2, x_3) = (3, 1, 7)$ yields the split-point (2, 2, 4) for step 1. Thus the first step of the Z_s-address consists of the three bits 101_2. The next split-point (1, 1, 2) compared to the modified $p' = (1, 1, 3)$ yields 111_2. At this step the resolution of dimension 2 and 3 is exhausted. Thus the third step just consists of one dimension (i.e., one bit). The split-point (1) compared to the modified and now one-dimensional tuple $p'' = (1)$ results in 1_2 for step 3. The entire Z_s-address is Z-address(p) = $101_2.111_2.1_{2s} = 5.7.1_s$.

Definition 3-27 (contribution of a dimension to a Z_s-address): The contribution $\text{contrib}_i(\alpha)$ of dimension i to a Z_s-address α is:

$$\text{contrib}_i(\alpha) = \sum_{j=1}^{\text{steps}(i)} \alpha_{j,i} \cdot 2^{\text{steps}(i)-j}$$

Example 3-7:

The contributions for α = Z-address(p) of Example 3-6 are $\text{contrib}_1(\alpha) = 11_2 = 3$, $\text{contrib}_2(\alpha) = 01_2 = 1$ and $\text{contrib}_3(\alpha) = 111_2 = 7$.

Lemma 3-19: For Z_s-addresses the contribution of an attribute is just the attribute itself.

Proof:

According to Algorithm 3-1 bit c_j of the contribution $c = \text{contrib}_i(\alpha) = c_{\text{steps}(i)-1}...c_0$ represents a binary decision between less than or equal or greater than with respect to the split-point $2^{\text{steps}(i)-1-j}$. Summing up all bits of the contribution from right to left and weighting each bit j with the value of its position 2^j then results in attribute A_i with $x_i = \text{cartesian}(\alpha)_i$. □

However, in Section 5.3.5 we will describe a variant of the address calculation, where $\text{contrib}_i(\alpha) \neq x_i$.

Lemma 3-20 (domain of a step of a standard address): Z_s-addresses result in a subcube numbering between 0 and 2^d-1.

Proof:

Each subdivision step of the Z_s-address calculation consists of d binary decisions, one for each dimension. Since each decision divides the space in the middle, after d decisions the space got divided in every dimension. We thus have obtained a subcube of the multidimensional space. By storing the binary decision of each dimension in a bit, we obtain a bit string with d bits. Each combination of bits denotes a different subcube. With d bits 2^d subcubes numbered from 0 to 2^d-1 exist. □

Lemma 3-21 (length of a Z_s-address): The length in bits of Z_s-addresses is identical for each tuple of a given universe. If the domain of dimension i consists of r_i distinct values, this length is calculated as:[11]

$$\text{addresslength}(\Omega) = \sum_{i=1}^{d} \text{steps}(i) = \sum_{j=1}^{\text{steps}(1)} \text{steplength}(j)$$

Proof:

For each dimension Algorithm 3-1 divides the domain in the middle until the limit of resolution is reached. Each of these subdivisions is represented by one bit. To reach the limit of resolution for a domain consisting of r_i distinct values, steps(i) = $\log_2 r_i$ subdivisions are necessary. Doing this for all dimensions results in $\sum_{i=1}^{d} \text{steps}(i)$ subdivisions, each of which is reflected by one bit of the address. This proves the first part of the equation. According to the definition of Z_s-addresses the second part of the formula describes the length in bits as the sum of the length of the steps of the address in bits. □

Example 3-8:

The 3d-universe of Example 3-6 with $r_1 = 4$, $r_2 = 4$, $r_3 = 8$ yields the number of bits for each dimension: $\log_2 r_1 = 2$, $\log_2 r_2 = 2$ and $\log_2 r_3 = 3$. Thus Z_s-addresses for this universe have a length of 7 bits.

Definition 3-28 (address incrementation and decrementation): For a Z-address α we define the Z-addresses $\alpha \oplus 1$ (and $\alpha \ominus 1$) as follows: Consider α as a bit string and add to α (or subtract from α) the binary number 1. The resulting bit string is then split into steps again and thereby defines the incremented Z-address $\alpha \oplus 1$ (or decremented Z-address $\alpha \ominus 1$).

Example 3-9:

Incrementing and decrementing α = Z-address(p) = $5.7.1_s$ of Example 3-6 yields:

- $\alpha \oplus 1 = 5.7.1_s \oplus 1 = 1011111_2 + 1 = 1100000_2 = 6.0.0_s$
- $\alpha \ominus 1 = 5.7.1_s \ominus 1 = 1011111_2 - 1 = 1011110_2 = 5.7.0_s$

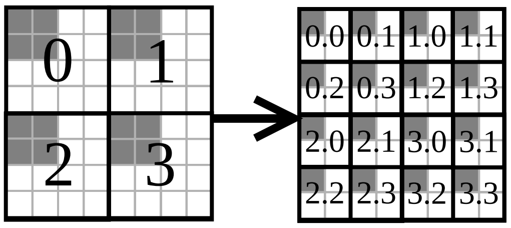
[11] Since Z-addresses are calculated with dimensions sorted in descending order of their resolution, steps(1) denotes the number of steps of a Z-address.

Theorem 3-4 (isomorphism between Z_s-addresses and Z_i-addresses): For Z-addresses $\neq \varepsilon_i$ the standard representation and the increment representation of Z-addresses are isomorphic.

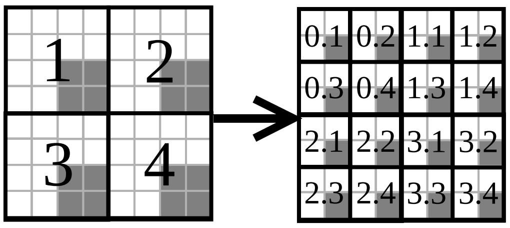
Proof:

For this proof it is important to remember that Z_i-addresses can be enlarged to the length of Z_s-addresses (Lemma 3-16). Since there exists no counterpart to ε_i in Z_s-addresses, we must exclude ε_i from the proof. The subcubes of a Z-area defined by a Z_i-address are then numbered from 1 to 2^d, whereas the subcubes of Z_s-addresses are numbered from 0 to 2^d-1. The fundamental difference between standard representation and increment representation is that, with respect to further subdivision steps, in standard representation a subcube number denotes the upper left corner of a subcube, whereas the lower right corner of the subcube is denoted in increment representation. In order to address points which are (in Z-ordering) less than the lower right corner, the subcube number has to be decremented by 1 in increment representation. Therefore the domain of the subcube numbers of Z_i-addresses must be enhanced by an additional subcube number zero for each but the last subdivision step (at the limit of resolution no further subdivisions take place, thus a step number of zero is not possible for the last step of a Z_i-address).This is illustrated in Figure 3-9.

Considering the area containment up to the last subdivision step there is no difference between standard representation and increment representation, since by adding the additional subcube number zero both representations denote the same subcube by the same number. Because of the different subcube numberings at the limit of resolution (i.e., for the last step) the subcube number in increment representation is by one larger than the subcube number in standard representation. □



(a) standard representation



(b) increment representation

Figure 3-9: Address representation

The proof of Theorem 3-4 immediately yields a bijective function to switch between Z_i-addresses and Z_s-addresses: If we enlarge the Z_i-address to the length of the Z_s-address and write it as a binary string:

$$\alpha_{\text{increment}} = \alpha_{\text{standard}} \oplus 1.$$

In the same way, we calculate:

$$\alpha_{\text{standard}} = \alpha_{\text{increment}} \ominus 1.$$

Note that in increment representation the very first step consists of d+1 bits, since it must be possible to represent the number 2^d in this step.

Example 3-10:

The point p = (3,1,7) of Example 3-6 has the Z_s-address $5.7.1_s$, which yields the binary string $(0)1011111_2$. Binary incrementing this Z_s-address by 1, yields the string $(0)1100000_2$. Splitting this binary string in steps yields $(0)110_2.000_2.0_2$. Removing trailing zeroes yields the Z_i-address 6_i.

Example 3-11:

In the same way we calculate the Z_s-addresses for the Z-areas area($0.0.1_i$), area($1.3.2_i$), area(2.1_i) , area(3_i) and area(4_i) of the two-dimensional universe with resolution 8 of Figure 3-8:

Z_i-address	bitstring	decremented bits	splitting in steps	Z_s-address
$0.0.1_i$	$(0)000001_2$	$(0)000000_2$	$(0)00_2.00_2.00_2$	$0.0.0_s$
$1.3.2_i$	$(0)011110_2$	$(0)011101_2$	$(0)01_2.11_2.01_2$	$1.3.1_s$
$2.1.0_i$	$(0)100100_2$	$(0)100011_2$	$(0)10_2.00_2.11_2$	$2.0.3_s$
$3.0.0_i$	$(0)110000_2$	$(0)101111_2$	$(0)10_2.11_2.11_2$	$2.3.3_s$
$4.0.0_i$	$(1)000000_2$	$(0)111111_2$	$(0)11_2.11_2.11_2$	$3.3.3_s$

Table 3-1: Transformation from Z_i-addresses to Z_s-addresses

Lemma 3-22 (open regions and closed regions): $]\,\alpha : \beta\,] = [\,\alpha \oplus 1 : \beta\,]$

Proof:

$]\,\alpha : \beta\,]$ denotes the space area(β) \ area(α). This is the subspace of multidimensional space covered by the Z-address interval $]\,\alpha, \beta\,]$. The point corresponding to the Z-address $\alpha \oplus 1$ is included in $]\,\alpha : \beta\,]$, i.e., $]\,\alpha, \beta] = [\alpha \oplus 1, \beta\,]$. Thus the Z-region $[\alpha \oplus 1 : \beta\,]$ as defined in Section 3.4 is identical to the Z-region $]\,\alpha : \beta\,]$ as defined in this section. □

Lemma 3-23: The ordering $\prec$ of Z-values of Section 3.1 is identical to the ordering $\prec^*$ defined by Algorithm 3-1.

Proof:

Algorithm 3-1 calculates the ordinal numbers for $\prec$ by storing the result of a binary decision between $x_i' < 2^{d-j-1}$ and $x_i' \geq 2^{d-j-1}$ for each dimension $i \in D$ in bit $\alpha_{j \cdot d+i-1}$ of the binary representation of the address $\alpha = \alpha_{s \cdot d-1} \ldots \alpha_0$. According to Definition 3-3 Z(x) is calculated exactly in the same way. Therefore Z-values are merely another interpretation of the addresses calculated by Algorithm 3-1. □

3.6.2 Volumes of Z-Areas and Z-Regions

The volume of a Z-area Λ is the percentage of the entire space that is covered by Λ. Since area(ε_i)= $\emptyset$, vol(area(ε_i)) = 0. If a Z-area has a positive volume, this volume can easily be calculated from the Z_s-address of the Z-area:

Lemma 3-24: If $\beta_1 ... \beta_k$ is the bit-sequence of a Z_s-address numbered from left to right, the volume of the corresponding area is calculated as

$$vol(area(\beta_1 ... \beta_k)) = 2^{-k} + \sum_{i=1}^{k} \beta_i \cdot \frac{1}{2^i}$$

Proof:

According to the definition of Z_s-addresses, each bit β_i of a standard address represents a decision between two parts of space with equal volume. If β_i is set, the first half of the space is completely contained in the area. Otherwise, only a part of the first half of the space is contained in the area. This part is described by the following bits of the address. Since each subdivision step divides the previous subspace into two spaces of equal volume, β_i describes a binary decision in a subspace with a volume of 2^{-i}: If β_i is set, a subspace with volume 2^{-i} is included in the area. Thus $\beta_i \cdot 2^{-i}$ describes the contribution of the subspace of the volume 2^{-i} to the overall volume of area(β). At the limit of resolution either one or two subcubes with volume 2^{-k} are included. If β_k is zero, only the first subcube is included, otherwise both subcubes are included. We immediately get the above formula by weighting each set bit by the volume of the corresponding subspace and adding up these volumes and correctively adding 2^{-k} for the last step. □

Example 3-12:

The volumes of the Z-areas in Figure 3-8 are:

- vol(area(ε_i)) = 0
- vol(area($0.0.1_i$)) = vol(area($00_2.00_2.00_{2s}$)) = 1/64
- vol(area($1.3.2_i$)) = vol(area($01_2.11_2.01_{2s}$)) = ¼+ 1/8 + 1/16 + 1/64 + 1/64 = 30/64
- vol(area(2.1_i)) = vol(area($10_2.00_2.11_{2s}$)) = ½ + 1/32 + 1/64 + 1/64 = 36/64
- vol(area(3_i)) = vol(area($10_2.11_2.11_{2s}$)) = ½+1/8+1/16+1/32+1/64+1/64= 48/64
- vol(area(4_i)) = vol(area($11_2.11_2.11_{2s}$)) = ½+1/4+1/8+1/16+1/32+1/64+1/64 = 1

Lemma 3-25: If $\alpha_1 ... \alpha_k$ and $\beta_1 ... \beta_k$ are the bit sequences of the addresses α and β, the volume of the Z-region $]\alpha : \beta]$ is calculated as

$$\mathrm{vol}(]\alpha : \beta]) = \mathrm{vol}(\mathrm{area}(\beta)) - \mathrm{vol}(\mathrm{area}(\alpha)) = \sum_{i=1}^{k} (\beta_i - \alpha_i) \frac{1}{2^i}$$

Proof:

$]\alpha : \beta]$ = area(β) \ area(α). Applying Lemma 2-3 and Lemma 3-24 immediately yields the proof. □

Example 3-13:

The volumes of the Z-regions in Figure 3-8 are:

- vol($]\varepsilon_i : 0.0.1_i]$) = vol(area($0.0.1_i$)) – vol(area($\varepsilon_i$)) = 1/64 – 0 = 1/64
- vol($]0.0.1_i : 1.3.2_i]$) = vol(area($1.3.2_i$)) – vol area($0.0.1_i$) = 30/64 – 1/64 = 29/64
- vol($]1.3.2_i : 2.1_i]$) = 6/64
- vol($]2.1_i : 3_i]$) = 12/64
- vol($]3_i : 4_i]$ = 16/64

3.7 Independent Dimensions

In the following we consider multidimensional universes which are partitioned into Z-regions by point data which is distributed independently in each dimension.

Definition 3-29 (dependence and independence of dimensions): We call two dimensions x_i and x_j of a multidimensional data distribution *dependent*, if there exists a mapping f such that $f(x_i) = x_j$. Otherwise we call x_i and x_j *independent*.

In the following P(x) is the probability of the occurrence of a tuple $x \in \Omega$ in the relation R. By $\mathrm{P}_i(x_i)$, $i \in D$, we denote the probability of the occurrence of the attribute value $x_i \in \{1,...,r_i\}$. By $\mathrm{F}_i(x_i)$ we denote the cumulated distribution for the attribute value $x_i \in \{1,...,r_i\}$ of dimension i, i.e.,

$$\mathrm{F}_i(x_i) = \sum_{c=1}^{x_i} \mathrm{P}_i(c)$$

For independent data distributions of each attribute the probability $\mathrm{P}(x_1,...,x_d))$ is the product of the probabilities $\mathrm{P}_i(x_i)$, i.e.,

$$\mathrm{P}(x_1,..., x_d) = \prod_{i=1}^{d} \mathrm{P}_i(x_i)$$

In the following we consider several cases of independent data distributions for each dimension, namely uniformly distributed data, Gaussian distributed data and combinations of both.

3.7.1 Uniformly Distributed Data

The most simple data distribution for a domain is uniformly distributed data. For *uniformly distributed data* the probability of the occurence of a specific value is constant and identical for each value, i.e.,

$$P_i(x_i) = 1/|\mathbb{D}_i|$$

Uniform distribution is in general only an approximation for computer generated data. Real-world applications seldom create uniformly distributed data. Nevertheless, this type of distribution is interesting from a theoretical point of view: If the data distribution of an attribute is unknown, it is often useful to assume uniform distribution in order to not disfavor a certain value. In addition the theoretical analysis is greatly simplified by the assumption of uniformly distributed data.

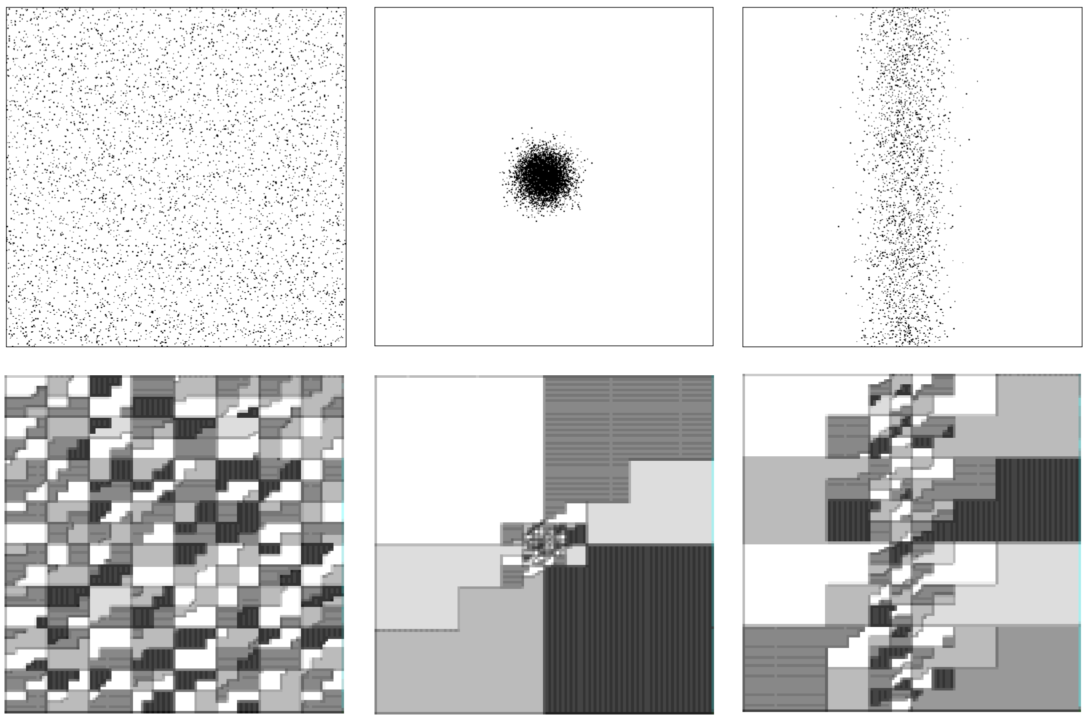


Figure 3-10: Uniformly and Gaussian distributed data

Figure 3-10a shows the point and Z-region distribution for a partitioned relation with 5000 points stored on about 200 disk pages (i.e., about 200 Z-regions), where the values of both dimensions are distributed uniformly in the corresponding domain.

3.7.2 Gaussian Distributed Data

A very frequent data distribution for practical applications is the *Gaussian distribution* (or normal distribution). The distribution function of the Gaussian distribution is:

$$\mathrm{P}_i(x_i) = \frac{1}{\sigma \cdot \sqrt{2 \cdot \pi}} \cdot e^{-\frac{(x_i-\mu)^2}{2 \cdot \sigma^2}}$$

$$\mathrm{F}_i(x_i) = \Phi_{\mu,\sigma}(x_i) = \frac{1}{\sigma \cdot \sqrt{2 \cdot \pi}} \cdot \int_{-\infty}^{x_i} e^{-\frac{(u-\mu)^2}{2 \cdot \sigma^2}} du$$

Gaussian distributed data is located around the average value μ with a standard deviation of σ. Therefore the data is clustered around the center μ. Figure 3-10b shows a multidimensional data distribution, where both dimensions are independently Gaussian distributed, and the corresponding region partitioning of the multidimensional space. Figure 3-10c shows a multidimensional data distribution, where the horizontal dimension is Gaussian distributed and the vertical dimension is uniformly distributed.

These pictures show an underlying principle of Z-region partitioning: Z-region partitioning adapts to the density of the data, while trying to preserve locality in the partitions (=regions).

3.7.3 Idealized Uniform Partitioning

In this section we investigate a special case of uniformly distributed data, namely an idealized uniformly partitioned universe.

Definition 3-30 (idealized uniform partitioning): A set of P addresses is *idealized uniformly distributed*, if all of these addresses only differ in the $\lceil \log_2 P \rceil$ leftmost bits, whereby all possible combinations of the $\lfloor \log_2 P \rfloor$ leftmost bits exist and all bits with a position greater than $\lceil \log_2 P \rceil$ are set. We call the partitioning of the multidimensional space introduced by the sequence of these Z-addresses *idealized uniform partitioning*.

An idealized uniform partitioning produces rectangular regions, where each region has either the shape of a subspace with volume $2^{-\lceil \log_2 P \rceil}$ or consists of two of these subspaces. An idealized uniform partitioning for the data of Figure 3-11a is illustrated in Figure 3-11b. If further data is inserted, the idealized uniform partitioning consists of even more quadratic Z-regions (Figure 3-11c). This continues, until $\log_2 P = \lceil \log_2 P \rceil$. Then the partitioning consists of a complete “chessboard”; further insertion then halves the quadratic regions.

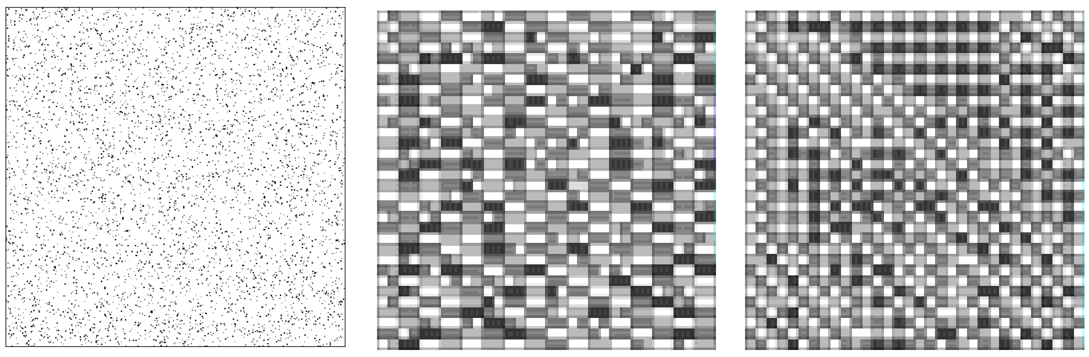

(a) (b) (c)

Figure 3-11: Idealized uniform partitioning

Independent and uniformly distributed key attributes are not very common in practice. However, the concept of Variable UB-Trees as described in Section 5.3.5 allows to create a uniform partitioning of independently distributed dimensions regardless of their data distribution. Many of the observations derived for uniformly partitioned data also hold for Variable UB-Trees.

3.8 Dependent Dimensions

Dependencies or correlations between attributes often occur in practical applications. The salary, for instance, is very often related to the age of a person, the horse powers of a car's engine are normally related to the price of the car. In the following sections we investigate several types of dependent data distributions.

3.8.1 Linear Dependency

Definition 3-31 (linear dependency): Two dimensions *i* and *j* are *linearly dependent*, if the value of x_i is a linear function of the value of x_j, i.e., for $m \neq 0$ and an arbitrary value $c \in \mathbb{N}_0$:

$$x_i = m \cdot x_j + c$$

Linearly dependent dimensions can cause a dependence of particular bits of the values b_i of x_i and b_j of x_j. This restricts the number of bit combinations possible for addresses and therefore reduces the number of splits that are necessary to entirely subdivide the universe.

Figure 3-12a and Figure 3-12b show two special cases of linear dependency, namely the identity (m = 1, c = 0) and the inverse identity (m = -1, c = 0).

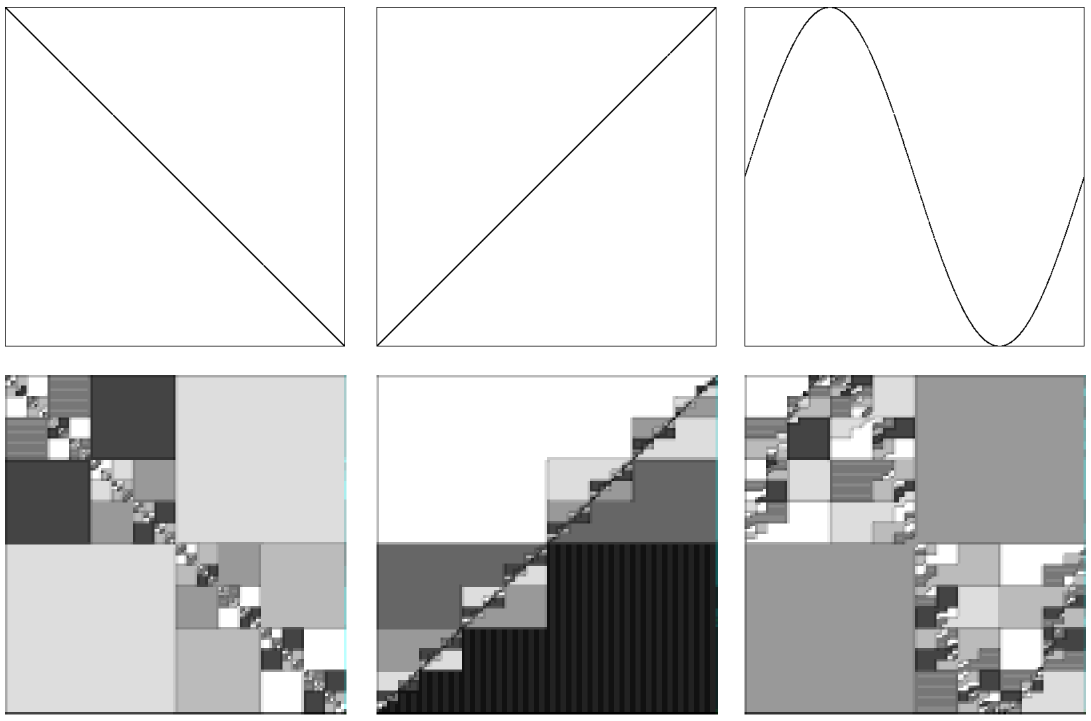


Figure 3-12: Dependent dimensions

3.8.2 Further Dependencies

We use sine dependency to illustrate the behavior of Z-region splits for non-linear dependencies. Sine dependencies occur, if one dimension is periodically growing and shrinking. This is true for construction workers during summer and winter, for stocks rates, etc.

Definition 3-32 (sine dependency): Two dimensions *i* and *j* are *sine dependent*, if the value of x_i can be calculated by a sine function of the value of x_j, i.e., for $f \neq 0$, $a \neq 0$:

$$x_i = a \cdot \sin f \cdot x_j$$

As with linear dependencies sine dependency causes correlations between some bits of the attributes. This again means that one split of a Z-region partitions the universe with respect to several attributes. For more complicated dependencies this effect may not be as powerful as for the simple cases of sine dependency and linear dependency, since it might not occur globally in the entire universe, but locally constrained to some part of the universe.

Figure 3-12c shows sine dependency and the corresponding Z-region partitioning for $a = 1$ and $f = 1$.

3.9 High Dimensionalities

Preserving spatial proximity becomes increasingly difficult with increasing dimensionality. In *d*-dimensional space each point that is not situated on some border of the universe has $2 \cdot d$ neighbors. When recursively partitioning a *d*-dimensional space *s* times in each dimension, $2^{s \cdot d}$ partitions are created. Since the Z-regions of a Z-region partitioning correspond to database pages, each additional partitioning level requires an exponential increase in the database size.

Definition 3-33 (total split depth; ideal split depth per dimension): We call the number of completed recursive splits of a Z-region partitioning its *total split depth* $l_{\downarrow}$. For a Z-region partitioning consisting of *P* idealized uniformly distributed Z-regions, the total split depth is calculated as:

$$l_{\downarrow}(P) = \lfloor \log_2 P \rfloor$$

$\lfloor l_{\downarrow}(P)/d \rfloor$ then is the lower bound for the *ideal split depth per dimension*. The rightmost[12] $l_{\downarrow}(P)$ mod *d* dimensions have one additional completed split level. Thus the upper bound of the ideal split depth per dimension is $\lfloor l_{\downarrow}(P)/d \rfloor + 1$.Therefore we can calculate the *ideal split depth* for dimension *i* as:

$$l_i(P) = \begin{cases} \left\lfloor {}^{l_{\downarrow}(P)}\!/_{d} \right\rfloor + 1 & , l_{\downarrow}(P) \bmod d > d - i \\ \left\lfloor {}^{l_{\downarrow}(P)}\!/_{d} \right\rfloor & , \text{otherwise} \end{cases}$$

Figure 3-13 displays the ideal split depth per dimension for Z-region partitionings with 1 up to 20 dimensions. The figure shows that dimensionalities larger than 6 result in an ideal split depth of at most 5 for tables of 80 GB. However, an ideal split depth of 5 means $2^5 = 32$ subdivisions in each dimension. Dimensionalities larger than 12 never exceed an ideal split depth of 3 for even huge databases. According to the Figure dimensionalities larger than 20 almost forever consist of only one split level for each dimension and for dimensionalities larger than 32 one can expect that many of these dimensions will not be split at all and thus do not contribute to the space partitioning.

This means that for average to large sized tables the Z-region partitioning yields a suitable space partitioning for dimensionalities less than or equal to 6. The multidimensional space partitioning deteriorates exponentially with increasing dimensionality, since the table size needs to grow exponentially to compensate an increase in dimensionality.

[12] We have defined the address calculation to use the order *d*, *d*-1, *d*-2, ..., 2, 1 of dimensions to calculate addresses. Therefore the rightmost dimensions are subdivided first.

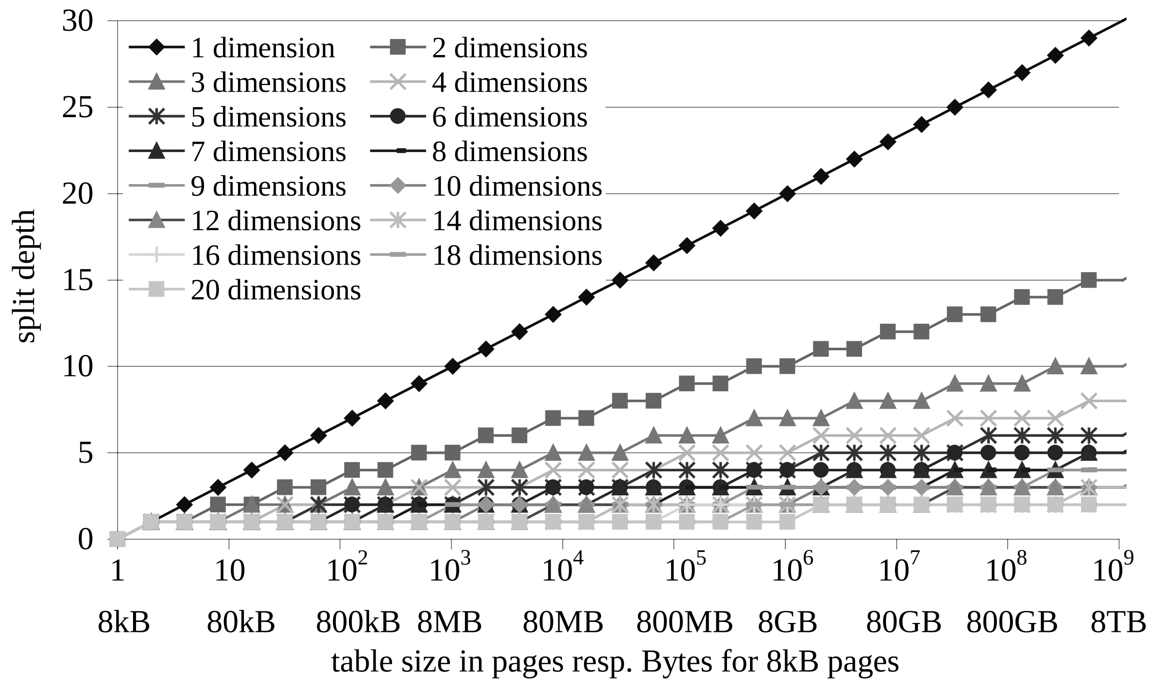


Figure 3-13: Split depth per dimension for idealized uniformly distributed regions

3.10 Utilization of the multidimensional Space

Definition 3-34 (actual domain of a dimension): For a relation $R(x_1,...,x_d)$, $x_i \in \mathbb{D}_i$ for all $i \in D$, the *actual domain* $\mathbb{V}_i$ of dimension i is:

$$\mathbb{V}_i = \{x_i \in \mathbb{D}_i \mid (x_1, ..., x_{i-1}, x_i, x_{i+1}, ..., x_d) \in R\}$$

In most cases, the actual domain of an attribute and the data distribution in that domain are not known in advance. In real world applications, usually $\mathbb{V}_i \subsetneq \mathbb{D}_i$, since not all anticipated values exist in the database. On the other hand one easily runs into problems when the specified domain is too small, e.g., the currently very much discussed year-2000-problem, where $\mathbb{V}_i = [0, 99]$ and soon $\mathbb{V}_i \supset \mathbb{D}_i$.

Definition 3-35 (partition of an actual domain): For $i \in D$, $s \in \{0, 1, ..., \log_2 r_i\}$ and $k \in \{1,...,2^s\}$ we define the *k,i,s-partition of the actual domain* $\mathbb{V}_i$ as the values of the actual $\mathbb{V}_i$ which are located between $(k\text{-}1)\cdot 2^{-s}$ % and $k\cdot 2^{-s}$ % of the domain $\mathbb{D}_i$.

$$\text{partition}_{i,s}(k) = \{x_i \in \mathbb{V}_i \mid \hat{x}_i \in](k\text{-}1)\cdot 2^{-s}, k\cdot 2^{-s}] \text{ for } x_i \in \mathbb{D}_i\}$$

Definition 3-36 (prefix length of a partition): The *prefix length of a partition of an actual domain* is the length of the common prefix of the partition in binary representation, i.e., the number of common bits of all values in the partition:

$$\text{prefix-length}_{i,s}(k) = |\text{common-prefix}(\text{partition}_{i,s}(k))|, \text{ if partition}_{i,s}(k) \neq \emptyset$$

Example 3-14:

For the domains $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{D}_3 = [0, 7]$ we assume the actual domains of the relation *R* to be

- $\mathbb{V}_1 = \{000_2,001_2,010_2,011_2,100_2,101_2,110_2,111_2\}$,
- $\mathbb{V}_2 = \{000_2,001_2,010_2,011_2\}$,
- $\mathbb{V}_3 = \{000_2,001_2\}$

This yields the following partitions:

- $\text{partition}_{1,0}(1) = \mathbb{V}_1$
- $\text{partition}_{1,1}(1) = \{000_2,001_2,010_2,011_2\}$, $\text{partition}_{1,1}(2) = \{100_2,101_2,110_2,111_2\}$
- $\text{partition}_{1,2}(1) = \{000_2,001_2\}$, $\text{partition}_{1,2}(2) = \{010_2,011_2\}$, $\text{partition}_{1,2}(3) = \{100_2,101_2\}$, $\text{partition}_{1,2}(4) = \{110_2,111_2\}$

- $\text{partition}_{2,0}(1) = \mathbb{V}_2$
- $\text{partition}_{2,1}(1) = \{000_2,001_2,010_2,011_2\} = \mathbb{V}_2$, $\text{partition}_{2,1}(2) = \varnothing$
- $\text{partition}_{2,2}(1) = \{000_2,001_2\}$, $\text{partition}_{2,2}(2) = \{010_2,011_2\}$, $\text{partition}_{2,2}(3) = \varnothing$, $\text{partition}_{2,2}(4) = \varnothing$

- $\text{partition}_{3,0}(1) = \mathbb{V}_3$
- $\text{partition}_{3,1}(1) = \{000_2,001_2\} = \mathbb{V}_3$, $\text{partition}_{3,1}(2) = \varnothing$
- $\text{partition}_{3,2}(1) = \{000_2,001_2\} = \mathbb{V}_3$, $\text{partition}_{3,2}(2) = \varnothing$, $\text{partition}_{3,2}(3) = \varnothing$, $\text{partition}_{3,2}(4) = \varnothing$

Thus while the space partitioning of the domain yields a full partitioning of the actual domain for dimension 1, the actual domain of dimension 2 is unevenly partitioned. The actual domain of dimension 3 is not partitioned at all. In this case we get the following binary prefix lengths:

- $\text{prefix-length}_{1,0}(1) = 0$, $\text{prefix-length}_{1,1}(k) = 1$ for $k \in \{1,2\}$ and $\text{prefix-length}_{1,2}(k) = 2$ for $k \in \{1,2,3,4\}$
- $\text{prefix-length}_{2,0}(1) = 1$, $\text{prefix-length}_{2,1}(1) = 1$ and $\text{prefix-length}_{2,2}(k)= 2$ for $k \in \{1,2\}$
- $\text{prefix-length}_{3,0}(1) = 2$, $\text{prefix-length}_{3,1}(1) = 2$ and $\text{prefix-length}_{3,2}(1) = 2$
- All other prefix lengths are undefined ($= \perp$)

Definition 3-37 (actual split depth): The *actual split depth* l^*_i for a dimension is the number of bits of A_i which are used for the space partitioning of the UB-Tree.

If for any fixed $i \in D$ and $s < \log_2 r_i$ $\text{prefix-length}_{i,s}(k)$ is either constant or undefined for every $k \in \{1,...,2^s\}$, the actual split depth for each dimension can be calculated by Algorithm 3-2.

```
Input: $p$                     : number of pages
       $d$                     : number of dimensions
       prefix_length$_{i,j}$   : length of the common prefix of the first
                                 $j$ bits of the actual domain of attribute $i$
       $r$                     : cardinality of the domain of each attribute
Output: $l_1, ..., l_d$        : actual split depths

$l_1^* = ... = l_d^* = 0$
$s = l_\downarrow(p)$
$i = 1$
$j = 0$
repeat
   if prefix_length$_{i,j}$ <= $j$ then
        $l_i^* := l_i^* + 1$
        $s := s - 1$
   end if
   if $i < d$ then
        $i := i + 1$
   else
        $i := 1$
        $j := j + 1$
   end if
until $s = 0$ or $j > \log_2 r$
```

Algorithm 3-2: Calculation of the actual split depth

Providing unused extra values at some border of the domain or not using all intermediate values causes an imperfect domain exhaustion for an attribute. Since the Z-region partitioning performs a recursive subdivision in the middle for each step, this may result in a lower actual split depth for some dimensions.

Example 3-15:

For $P = 32$ the actual split depths of Example 3-14 for $P = 32$ are $l_1^* = 3$, $l_2^* = 2$ and $l_3^* = 1$. This means that the first partitioning in dimension 3 takes place after three subdivision steps in dimension 1 and two subdivision steps in dimension 2. This yields exactly the same multidimensional ordering as the compound ordering $<_{x1 \circ x2 \circ x3}$. Moreover, if the table size is not large enough to split 6 times, i.e., $P < 32$, the partitioning in dimension 3 is not reflected by the Z-region partitioning at all.

Definition 3-38 (normalized actual split depth per dimension and normalized actual split depth): We define the *normalized actual split depth for dimension i* as:

$$l_i' = \begin{cases} \max\limits_{j \in D} l_j^* & , l_i^* = \log_2 |V_\mathrm{i}| \\ l_i^* & , \text{otherwise} \end{cases}$$

The *normalized actual split depth* is defined as:

$$l'_\downarrow = \sum_{j \in D} l_j'$$

The *normalized actual split depth* normalizes the actual split depth for universes of different resolutions per dimension. If one attribute is partitioned entirely, its split depth is increased to the maximum actual split depth over all dimensions. This yields identical normalized split depth, if all dimensions are partitioned completely.

Different actual split depths mean that the actual domains do not partition the universe evenly. This causes a layered multidimensional partitioning that reminds us of a puff pastry. We therefore call this effect of the incomplete domain utilization *puff pastry effect*.

Definition 3-39 (puff pastry, puff pastry degree): If for a Z-region partitioning of a relation R the normalized actual split depths are not identical for all dimensions $i \in D$, we call the partitioning to be a *puff pastry*. The degree of the puff pastry is measured as the normalized standard deviation of the normalized actual split depths:

,

$$\text{puff-pastry-degree}(R) = \text{std}(\{l_i'/l'{\downarrow} \mid i \in D\}) \cdot d$$

The puff pastry degree is a measure for the asymmetry of a Z-region partitioning. The normalized actual split depth l_i' is used to take dimensions with variable cardinality into account. The value of the puff pastry degree is a number between 0 and 1, whereby 0 means a perfect Z-region partitioning without any puff pastry. If splits only take place with respect to one dimension, the most extreme puff pastry degree of 1 is achieved.

Example 3-16:

For $P = 2^4 = 16$ the relation R of Example 3-14 has a puff pastry degree of 0,75, for $P = 2^5 = 32$ the puff pastry degree is 0.00. The data of Figure 3-10a and Figure 3-10b as well as the data of Figure 3-12a and Figure 3-12b also have a puff pastry degree of 0,00. Figure 3-10c has actual split depth of $l_1^* = 2$ and $l_1^* = 6$, resulting in a puff pastry degree of 0.5.

Figure 3-14 shows three examples of multidimensional space utilization. Figure 3-14a and Figure 3-14b show data, which is distributed uniformly in the vertical dimension and Gaussian with a very small standard deviation in the horizontal dimension. In Figure 3-14a the average is exactly the center of the universe. Thus, for 50% of the values in the horizontal dimension the leftmost bit is set and for 50% of the value the leftmost bit is cleared. This results in an actual split depth of 1 in the first dimension versus an actual split depth of 4 for the second dimension, yielding a puff pastry degree of 0.6. In Figure 3-14b the average is slightly shifted to the left. Now all values of the horizontal dimension have the same prefix. Therefore the split depth in the horizontal dimension is reduced to zero, i.e., no subdivision with respect to this dimension has taken place. Figure 3-14b therefore shows a real puff pastry with a puff pastry degree of 1.00. The Z-region partitioning of Figure 3-14c is filled with data of 5 two-dimensional Gaussian distributions with different average values. Here five puff pastries for five data clusters superimpose each other. Since four of these clusters are in different parts of the universe, which result in different prefixes for the first two bits of the Z-region addresses, the overall puff pastry degree is not as strong as in Figure 3-14b.

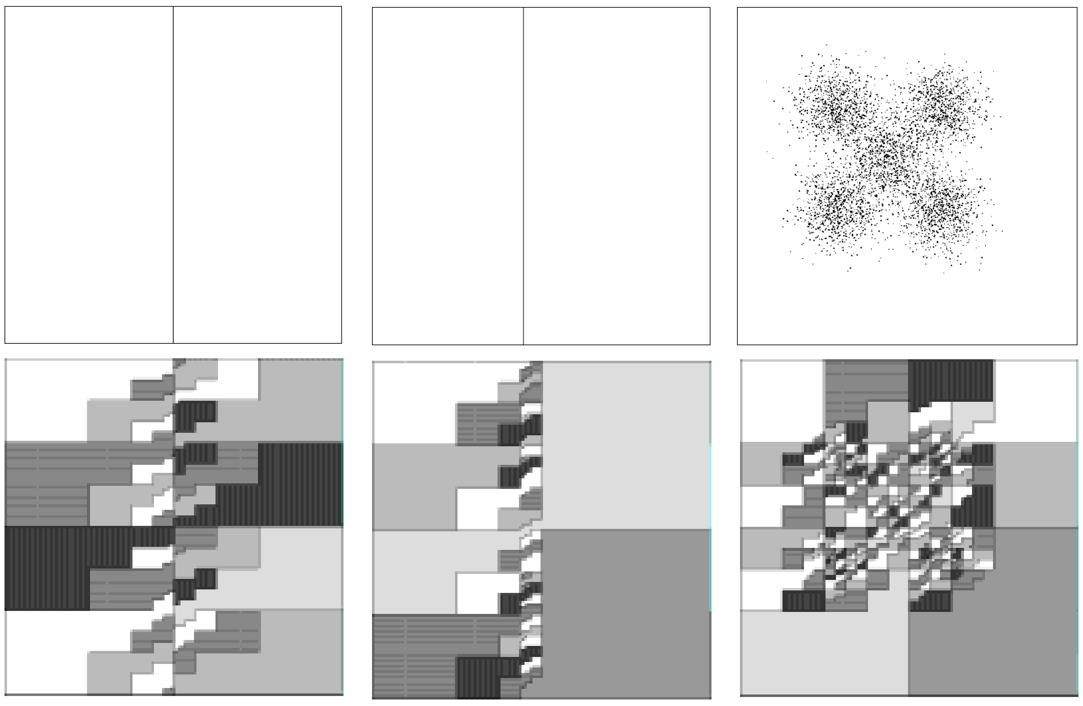


Figure 3-14: Utilization of the multidimensional space

In extreme cases as in Figure 3-14b the puff pastry effect results in a splitting similar to compound ordering. Thus the puff pastry effect seriously influences the multidimensional behavior of a Z-region partitioning and may in worst case totally destroy the multidimensional ordering. Therefore it is important to restrict domains as far as possible. Together with reorganization algorithms this strategy can avoid the puff pastry effect. If the data distribution of all dimensions is known in advance, the Z-region partitioning can take these data distributions into account and thereby avoid the puff pastry effect. This is achieved by the so-called *Variable UB-Tree*, which is described in more detail in Chapter 6.4.

Part II

Our Approach To Query Processing with Multidimensional Indexes

With information we can go anywhere in the world, we are like turtles, our houses always on our backs.

(John Le Carré)

Chapter 4

The UB-Tree

The totally ordered addresses of a region partitioning created by a space filling curve can be stored in any variant of a B-Tree. This allows to create a multidimensional index for a universe partitioned into regions. For our prototype implementation we use Z-region partitioning, which is implemented easily while showing beneficial properties for multidimensional clustering of tuples (see Chapter 3). First we introduce the basic concepts of the universal B-Tree (UB-Tree). Then we describe algorithms for insertion and deletion. The algorithm for exact match queries is derived from the corresponding algorithm for exact match queries in B-Trees. Range queries are performed by determining the smallest set of Z-regions of the multidimensional partitioned relation that builds a cover for the query box. The insertion, deletion, point-query, and range query algorithms were first described in [Bay96]. In addition to these algorithm we introduce two further algorithms in this thesis: Nearest neighbor queries are efficiently handled by the Spiral algorithm, a refinement of the range query algorithm which just retrieves Z-regions that store a nearest neighbor candidate. Our second algorithm is the Tetris algorithm, a technique to process a multidimensionally partitioned relation in the sort order of any attribute.

4.1 Concept of the UB-Tree

The UB-Tree [Bay96] uses a space filling curve to create a partitioning of a multidimensional universe while preserving multidimensional clustering. Using the Lebesgue-curve (Z-curve) it is a variant of the zkd-B-Tree [OM84]. The UB-Tree utilizes a B*-Tree to store the Z-addresses of a Z-region partitioning of the multidimensional space. Each Z-region is mapped onto one disk page. At insertion time a full Z-region $[\alpha : \beta]$ is split into two Z-regions by introducing a new Z-address γ with $\alpha \prec \gamma \prec \beta$. γ is chosen so that the first half (in Z-order) of the tuples stored on Z-region $[\alpha : \beta]$ is distributed to $[\alpha : \gamma]$ and the second half is stored on $[\gamma \oplus 1 : \beta]$. Thus a worst case storage utilization of 50% is guaranteed. There is some freedom of choice for the Z-region split. For optimal query performance the split algorithm for UB-Trees tries to maintain rectangular regions and minimize fringes whenever possible. The UB-Tree requires logarithmic time (in the cardinality of the partitioned relation) for the basic operations of insertion, point retrieval, and deletion.

We write page$(\alpha : \beta)$ for the page corresponding to the Z-region $[\alpha : \beta]$. Depending on the context we also use the notion *page* and notation *page*$(\alpha{:}\beta)$ for the set of tuples stored in Z-region $[\alpha : \beta]$. By count$(\alpha : \beta)$ we denote the number of objects located on page$(\alpha : \beta)$.

Definition 4-1 (UB-Tree): A UB-Tree is any variant of a B-Tree in which the keys are Z-addresses of Z-regions ordered by $\prec$. Since we get $\gamma = \beta \oplus 1$ for two neighboring Z-regions $[\alpha : \beta]$ and $[\gamma : \delta]$, it suffices to store the upper address of each Z-region in the B-Tree. Each leaf page holds the tuples belonging to the corresponding Z-region. For secondary UB-Trees only tuple identifiers are stored on the corresponding leaf page (see Figure 4-1).

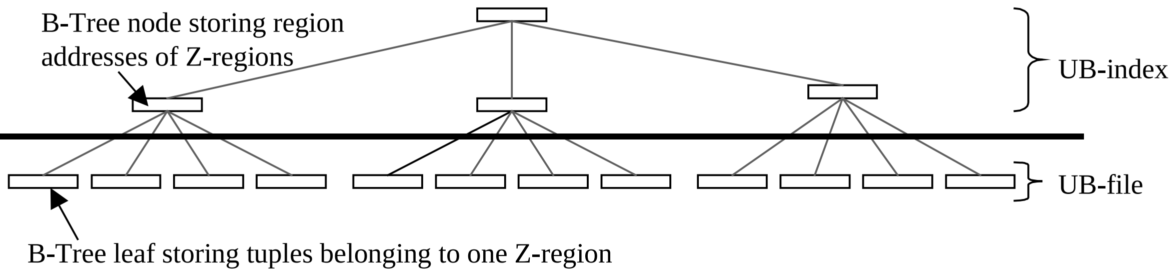

B-Tree node storing region addresses of Z-regions
UB-index
UB-file
B-Tree leaf storing tuples belonging to one Z-region

Figure 4-1: The UB-Tree

Example 4-1:

The five Z-regions in Figure 3-8 build a UB-Tree for the point data displayed in the lower right corner of Figure 3-8. Although Z-regions differ in volume, each Z-region stores about the same number of tuples because of the storage utilization guarantees of UB-Trees. Both the upper left corner and the lower right quarter of the universe contain five points, although the size (volume) of the region covering the lower right quarter of the universe is 16 times larger.

4.2 Insertion into UB-Trees

A tuple x to be inserted into the relation R is specified by its coordinates $(x_1, x_2, ..., x_d)$ with Z-address $\xi = Z(x_1, x_2, ..., x_d)$. x belongs to the unique Z-region $[\alpha{:}\gamma]$ satisfying $\alpha \preccurlyeq \xi \preccurlyeq \gamma$. Note that ξ must be computed only to a precision which is sufficient to determine the proper Z-region. x is inserted into the leaf-page corresponding to that Z-region, which is found by a point query. Since pages can store only a maximum number C of points, pages may overflow and be split like in B-trees. $[\alpha : \gamma]$ is split by introducing a new Z-area with Z-address β such that $\alpha \prec \beta \prec \gamma$. The Z-region $[\alpha : \gamma]$ is partitioned by β into $[\alpha : \beta]$ and $[\beta \oplus 1 : \gamma]$. The objects on page($\alpha : \gamma$) are distributed onto page($\alpha : \beta$) and page($\beta \oplus 1 : \gamma$) accordingly. β is constructed by increasing area(α) as follows: Add to area α subcubes from $[\alpha : \gamma]$ in increasing order until the number of the objects in $[\alpha : \beta]$ is between $½\,C - \varepsilon$ and $½\,C + \varepsilon$. If the next subcube in this process contains too many objects, it is recursively subdivided until the condition can be met. The parameter ε is used to get shorter split addresses, i.e., a better space partitioning. Section 5.4 shows how the parameter ε is used to reduce the number of Z-regions that overlap a query box and thus improves the performance of range queries in UB-Trees.

```
Input:  x : tuple to store in the UB-Tree
Output: none

$\xi$ = Z($x$)
find [$\alpha:\gamma$] in the UB-Tree, such that $\alpha \preccurlyeq \xi \preccurlyeq \gamma$
retrieve page($\alpha:\gamma$)
insert $x$ into page($\alpha:\gamma$)
if count($\alpha:\gamma$) > $C$
    choose $\beta \in [\alpha:\gamma]$, so that $½C - \varepsilon \leq$ count($\alpha:\beta$) $\leq ½C + \varepsilon$
    split page($\alpha:\gamma$) into page($\alpha:\beta$) and page($\beta \oplus 1 : \gamma$)
end if
```

Algorithm 4-1: Insertion algorithm for tuple x

The algorithm relies on the B-Tree operations to find the correct region. For n tuples stored in the database it inherits the I/O-complexity of $O(\log_C n)$. The CPU-complexity is also similar to that of B-Trees. The only difference is the address transformation, which is a very small amount of CPU time. We give performance figures for Z-address calculation in Section 5.3.1.

Figure 4-2 shows the insertion process into a UB-Tree. First the entire universe is stored on one disk page, the corresponding multidimensional space is represented by one Z-region (Figure 4-2a). After repeated insertion of tuples that page will overflow and is split. The multidimensional space is split into two Z-regions (Figure 4-2b). After further insertions the upper Z-region of Figure 4-2b indicated by the black arrow is split (Figure 4-2c). After further insertions an additional split takes place (Figure 4-2d). Figure 4-2e shows the universe after several further splits. The Z-region split in Figure 4-2f illustrates that splits in UB-Trees are local operations. Other regions are not affected by a split.

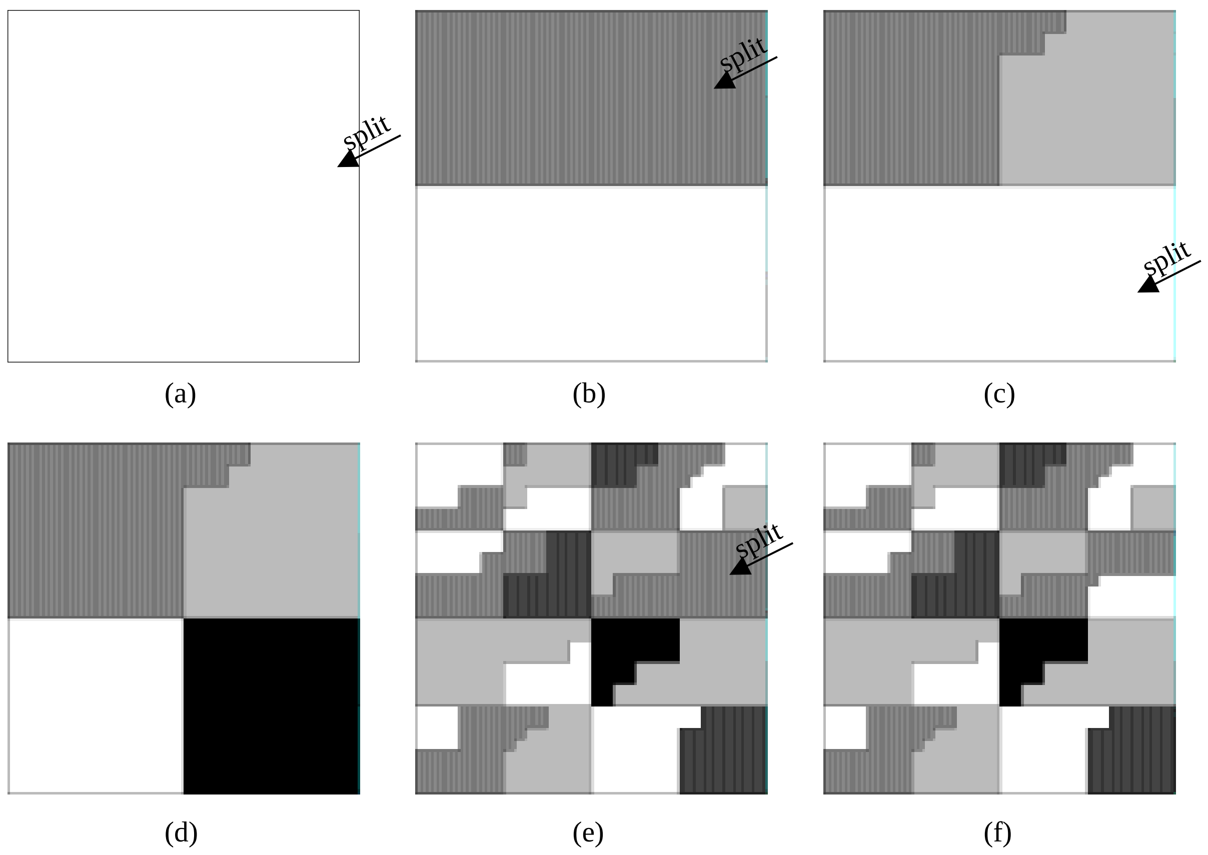


Figure 4-2: Insertion into UB-Trees

4.3 The Point Query Algorithm

To find a tuple $x = (x_1, x_2, ..., x_d)$ we compute its address $\xi := Z(x_1, x_2, ..., x_d)$ with sufficient precision to find the unique region $[\alpha : \beta]$ with the property $\alpha \preccurlyeq \xi \preccurlyeq \beta$ and fetch page($\alpha : \beta$) from the UB-file. This is achieved by searching the UB-index, using address ξ as the search key. page($\alpha : \beta$) must contain x with the additional information or the row id of x.

```
**Input:**  $x$ : tuple, only index attributes are specified
**Output:** $x$ : tuple, all attributes are specified

$\xi$= Z ($x$)
find [$\alpha$:$\beta$] in the UB-Tree, such that $\alpha \preccurlyeq \xi \preccurlyeq \beta$
retrieve page($\alpha$:$\beta$) into main memory
search content of page($\alpha$:$\beta$) to find $x$
```

Algorithm 4-2: Point query algorithm to find tuple *x*

This algorithm inherits the complexity of the underlying access structure for storing the addresses. The only additional overhead is the address calculation algorithm. An efficient implementation with a CPU-complexity linear in the tuple size is described in Section 5.3.

4.4 Deletion from UB-Trees

When tuples are deleted from $[\alpha:\beta]$, they are removed from page($\alpha:\beta$). If after this deletion page($\alpha:\beta$) contains less than ½ C - ε elements, then page($\alpha:\beta$) is merged with the following page($\beta\oplus 1:\gamma$) and the Z-region $[\beta\oplus 1:\gamma]$ disappears. If the resulting page($\alpha:\gamma$) overflows, it is split again "in the middle" by introducing a new Z-area with address δ and the regions $[\alpha:\delta]$ and $[\delta\oplus 1:\gamma]$ with the corresponding pages page($\alpha:\delta$) and page($\delta\oplus 1:\gamma$). This final split of regions and pages is analogous to the underflow technique between pages of B-trees [BM72].

```
Input:  x : tuple to delete from the UB-Tree
Output: none
// for easy illustration the algorithm does not handle the special
// case of the tuple being stored on the root page or last leaf

$\xi$ = Z(x)
search [$\alpha:\beta$]in the UB-Tree, such that $\alpha \preccurlyeq \xi \preccurlyeq \beta$
retrieve page($\alpha:\beta$)
delete x from page($\alpha:\beta$)
if count($\alpha:\beta$) < $½C-\varepsilon$
   merge page($\alpha:\beta$) with the neighboring page($\beta\oplus 1:\gamma$)
   into page($\alpha:\gamma$)
   if count($\alpha:\gamma$) > $C$
       choose $\delta\in[\alpha:\gamma]$ with count($\alpha:\delta$) $\le ½C-\varepsilon$
           and count($\delta\oplus 1:\gamma$) $\le ½C-\varepsilon$
       split page($\alpha:\gamma$) into page($\alpha:\delta$) and page($\delta\oplus 1:\gamma$)
   end if
end if
```

Algorithm 4-3: Deletion algorithm for a tuple *x*

The complexity considerations of insertion also apply to the deletion algorithm. For T tuples stored in the database the I/O-complexity is $O(\log_C T)$. The CPU-complexity is also similar to that of B-Trees.

4.5 The Range Query Algorithm

To answer a range query, only those Z-regions, which properly intersect the query box, must be fetched from the database and thus from the disk [Bay96]. Initially the range query algorithm calculates and retrieves the first Z-region that is overlapped by the query-box. Then the next intersecting Z-region is calculated and retrieved. This is repeated until a minimal cover for the query box has been constructed, i.e., the region that contains the ending point of the query box has been retrieved.

Definition 4-2 (subcube of an address): For any Z_i-address $\alpha = \alpha_1. ... .\alpha_k$ we define subcube(α), the hypercube-shaped multidimensional interval, which has $Z^{-1}(\alpha)$ as its end point and a normalized length of 2^{-k} in each dimension:

$$\text{subcube}(\alpha) = \{x \in \Omega \mid Z^{-1}(\alpha_1.\dots.(a_k - 1).\underbrace{0.\ \dots.0.1}_{r-k}) \trianglelefteq x \trianglelefteq Z^{-1}(\alpha_1.\dots.\alpha_k)\}$$

```
Input:  y,z : tuples that define a query box with $Z(y) \prec Z(z)$
Output: X   : result set of the range query

$\xi = Z(y)$; $\omega = Z(z)$; $X = \emptyset$
repeat
    find $[\alpha:\beta]$ in the UB-Tree, such that $\alpha \preccurlyeq \xi \preccurlyeq \beta$
    $X = X \cup \{(x_1,...,x_{d'}) \in [\alpha:\beta] \mid (x_1, ..., x_d) \in [[y, z]]\}$
    $\xi$ = Z-address of the first point intersecting the query box
    with $\xi \succ \beta$
until $\xi \succ \omega$
```

Algorithm 4-4: Range query algorithm for a query box $[[y, z]]$

It is important to note that for any Z-region the calculation of the next point intersecting the query box is performed solely in main memory. [Bay96] describes an algorithm that for a Z-region $[\alpha : \beta]$ and a query box Q calculates the largest Z-address $\gamma \in [\alpha : \beta]$ so that sub-cube(γ) $\cap$ $Q \neq \emptyset$. The algorithm then traverses all subcubes with greater Z-addresses, until one of these subcubes intersects Q. The smallest intersecting point p in Z-order then defines the next relevant Z-region, which is retrieved by a point query. Note that because of Lemma 3-16 the next subcube to subcube($\xi_1. ... .\xi_j.2^d$) is subcube ($\xi_1. ... .\xi_j$+1), which allows a more efficient traversal through the multidimensional space. However, the algorithm still has a CPU complexity which is exponential in the number of dimensions of the UB-Tree, which limits this algorithm to dimensionalities below 4. In Section 5.7.1 we sketch the implementation idea of a linear algorithm that solely operates on a binary representation of Z-addresses.

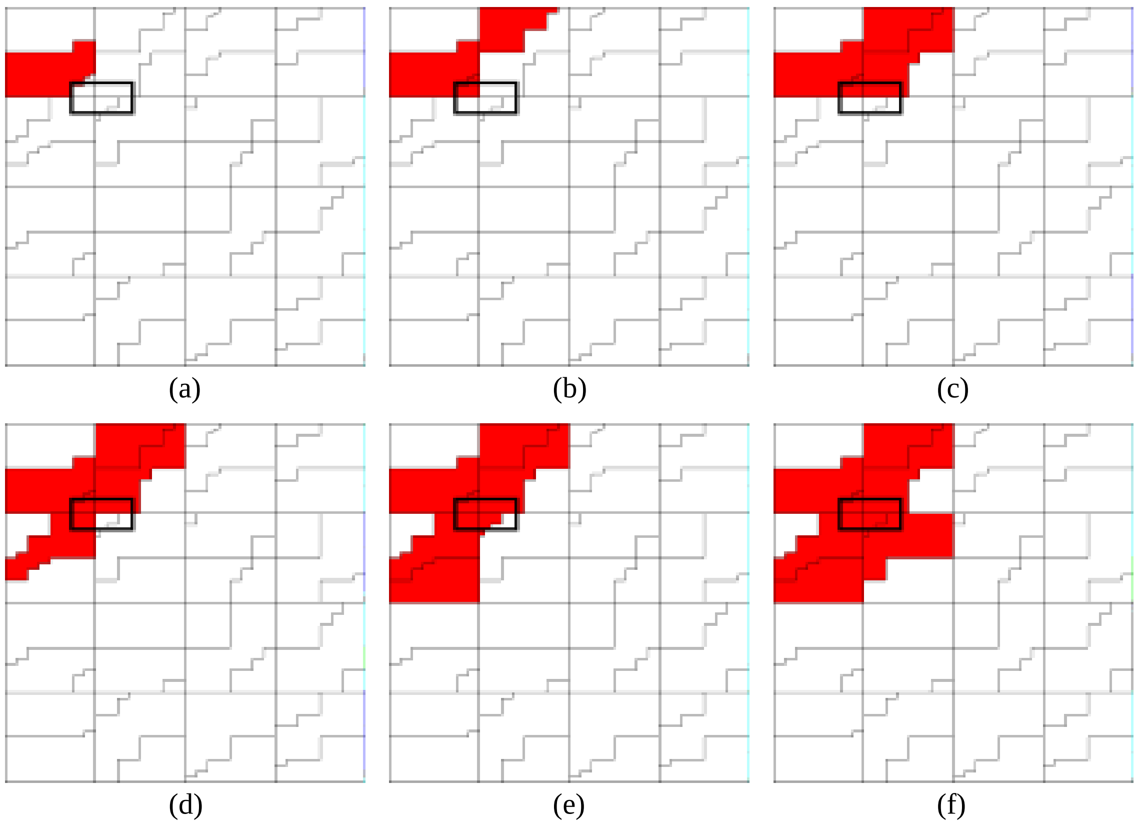


Figure 4-3: Processing a range query

Figure 4-3 illustrates the order in which the range query algorithm calculates and retrieves Z-regions for the range query illustrated by the black-bordered query box. For this example we use the notation $]\alpha : \beta]$ for $[\alpha \oplus 1 : \beta]$. The Z-regions overlapping the query box are shaded. Initially the algorithm uses the coordinate y of the query box $Q = [[y, z]]$ for a point query and thereby retrieves the Z-region $]0.0.0.1.3.2_i : 0.0.3.3.1.0.2_i]$ from the UB-index. Then the next Z-region that overlaps the query box must be determined (see also Figure 4-4a): The last subcube of $]0.0.0.1.3.2_i : 0.0.3.3.1.0.2_i]$ overlapping the query box is subcube($0.0.3.3.1_i$), which contains the point x_1 as the last intersecting point with respect to Z-ordering. The next subcube, subcube($0.0.3.3.2_i$), has an intersection with Q. The first intersecting point is p_1. A point query with p_1 yields the Z-region $]0.0.3.3.1.0.2_i : 0.1.1.1.0.2_i]$. Thus this Z-region may contain tuples, which are part of the result set of the range query. Finding the next Z-region works in the same way: The last intersecting subcube of $]0.0.3.3.1.0.2_i : 0.1.1.1.0.2_i]$ is subcube(0.1_i). The next intersecting subcube is subcube(0.2_i). The corresponding point query retrieves the Z-region $]0.1.1.1.0.2_i : 0.1.3.0.1_i]$, which again may contain tuples of the result set. For determining the next Z-region (see also Figure 4-4b) the algorithm starts with subcube($0.1.3_i$). subcube($0.1.4_i$) has no intersection with the query box. The next subcube with a larger Z-address is subcube(0.3_i), which has an intersection with the query box in p_2. The point query then retrieves the Z-region $]0.2.0.2.3.2_i : 0.2.2.1.1.2_i]$. Here we already see that the algorithm skipped Z-region $]0.1.3.0.1_i : 0.2.0.2.3.2_i]$, since this Z-region does not overlap the query box. In the same way the algorithm calculates and retrieves the Z-regions $]0.2.2.1.1.2_i : 0.3.0.0.2.3_i]$ and $]0.3.0.0.2.3_i : 0.3.2.1_i]$.

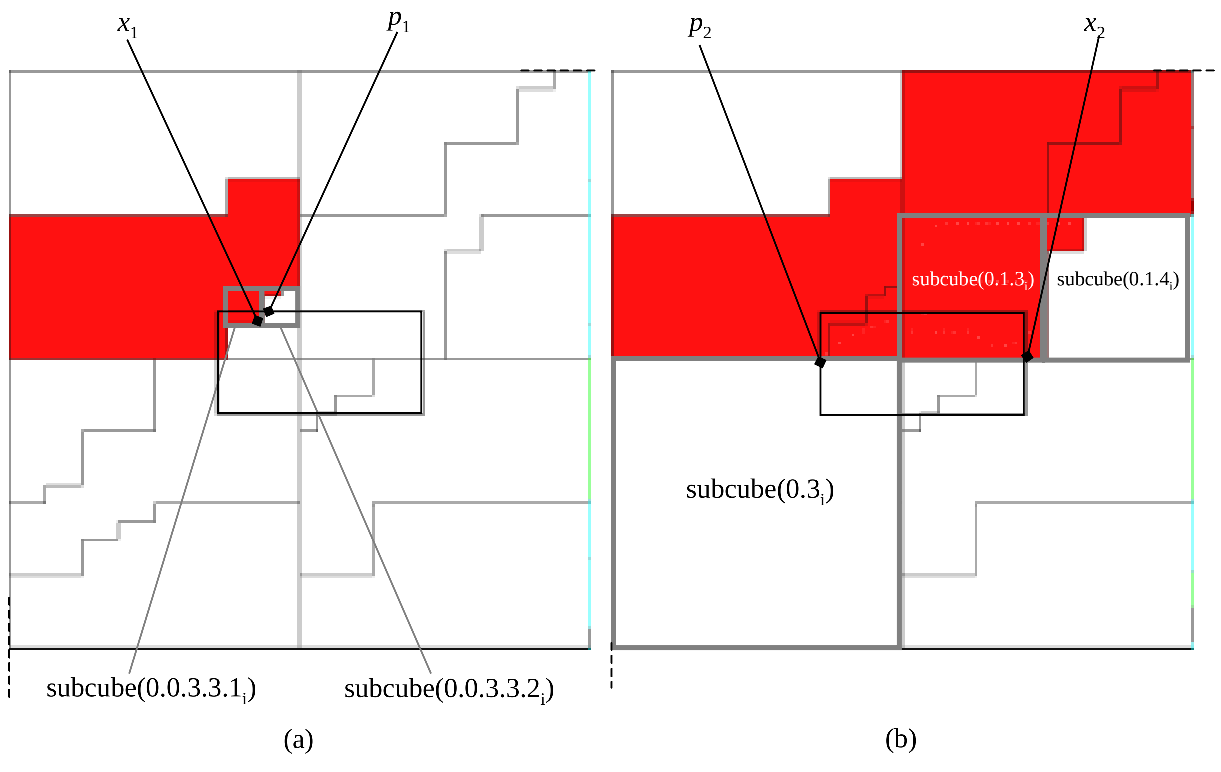


Figure 4-4: Zoom into Figure 4-3

The visualization already shows the principle benefit of the UB-Tree for multidimensional range queries: Only Z-regions that may contribute to the result set of a range query need to be retrieved from disk in order to answer the query. Thus in contrast to one-dimensional access methods as described in Section 2.4 the number of disk accesses of the UB-Tree range query algorithm is correlated to restrictions defined by all dimensions, not just to restriction in one dimension. We will analyze the theoretical and practical behavior of the range query algorithm in Chapters 6 and 7 and compare it to one-dimensional access methods.

Two strategies for returning the tuples of a query box are possible:

- *immediate tuple delivery*: As soon as a Z-region ρ is retrieved from the UB-index, the corresponding page(ρ) is retrieved from the UB-file. The tuples on page(ρ) are then extracted immediately and the tuples in the query box are returned. This strategy allows shorter response times and does not require any caching.
- *deferred tuple delivery*: After the upper Z-address of Z-region ρ has been retrieved from the UB-index, it is stored in a *region result set*. After all Z-addresses have been retrieved, the tuples of the entire region result set are materialized. This separation of UB-index access and UB-file access allows one to predict the tuple retrieval time before actually doing the retrieval. This may especially be useful for large result sets, where after retrieving the regions the user might be asked whether he really wants to materialize the result set.

Theorem 4-1 (range query theorem): The range query algorithm transforms the multidimensional interval [[y, z]] into a set of one-dimensional address intervals {]α, β],]β, γ], ...}, which with respect to the given region partitioning is the smallest cover for [[y, z]].

Proof: The proof is a direct consequence of Algorithm 4-4 and the fact that Z-regions are one-dimensional intervals. □

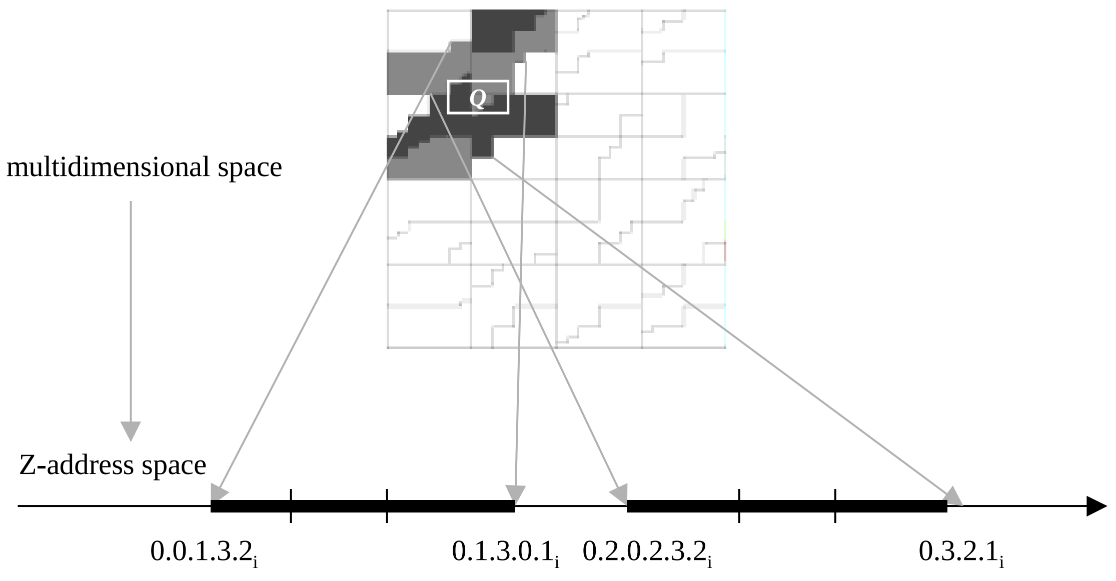


Figure 4-5: Transforming a query box into a set of Z-intervals

Very often some of these one-dimensional intervals are connected and therefore might be merged to larger one-dimensional intervals consisting of several regions. For instance, the six Z-regions overlapped by the query box of Figure 4-3 are six one-dimensional intervals, which could be merged into the two intervals $]0.0.1.3.2_i, 0.1.3.0.1_i]$ and $]0.2.0.2.3.2_i, 0.3.2.1_i]$, each of which consists of three Z-regions (see Figure 4-5). This merging could reduce the number of I/Os necessary to retrieve the pages of a query box, since for page clustered data only two random accesses are necessary versus six random accesses for tuple clustered data.

4.6 The Spiral Algorithm for Nearest-Neighbor Queries

For UB-Trees nearest neighbor queries can efficiently be processed in the following way: To find the nearest neighbor of point *x* we first retrieve the Z-region ρ where *x* would be located. Then we investigate all nearest neighbors to *x* that are stored on ρ. We have found a nearest neighbor, if the nearest neighbor *z* on ρ is closer to *x* than any point on the Z-region border of ρ. Otherwise we call *z* to be a nearest neighbor candidate. We now have to retrieve the closest not retrieved Z-region, since this Z-region might contain a point *z*' with a lower distance to *x* than *z*. This process must be iterated until no point on a border of the retrieved Z-regions has a distance to *x* less then the distance of the current nearest neighbor candidate.

We call this algorithm *spiral algorithm* because of the order that is used to retrieve the Z-regions in order to find the nearest neighbor.

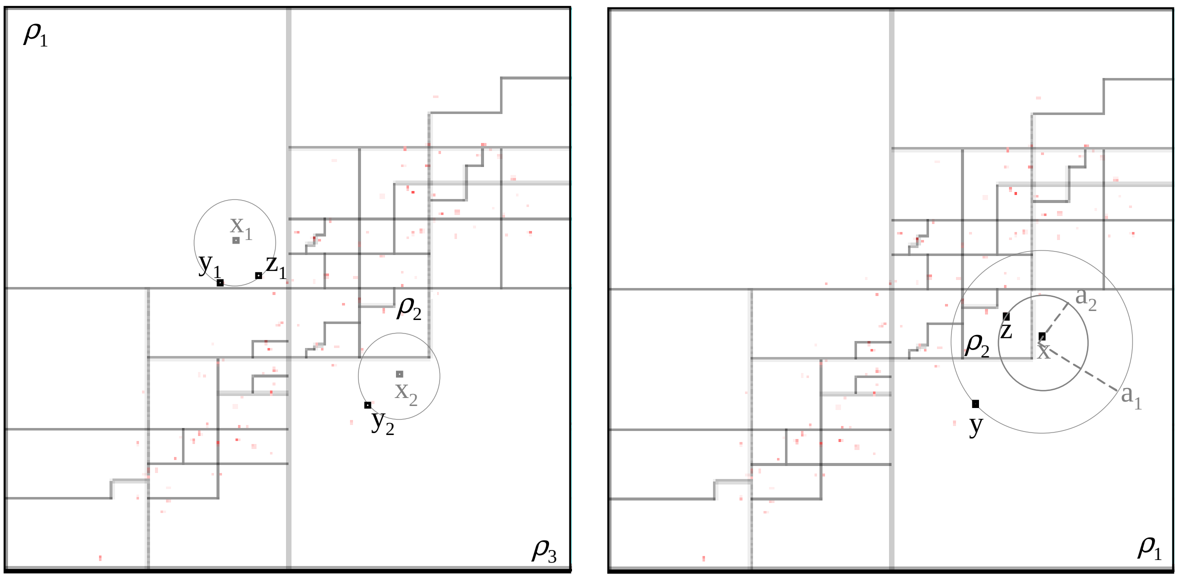
```
Input:  $x$ : tuple, for which the nearest neighbor is wanted
Output: $N$ : set of tuples containing the nearest neighbors to $x$

$\xi$ = Z($x$)
find [$\alpha$:$\beta$] in the UB-Tree, such that $\alpha \preccurlyeq \xi \preccurlyeq \beta$
$N$ = nearest neighbors of [$\alpha$:$\beta$]to x
$R$ = [$\alpha$:$\beta$]
while there is a point $y$ beyond the border of $R$ such that
        for some $z \in N$: distance($x$,$z$) > distance($x$,$y$)
        retrieve the Z-region [$\gamma$:$\delta$] bordering $R$,
            which has a point on its border
            that has the least distance to $x$
        $N$ = nearest neighbors of ([$\gamma$:$\delta$] $\cup$ $N$) to $x$
        $R$ = $R$ $\cup$ [$\gamma$:$\delta$]
end while
return the nearest neighbors of $N$ to $x$
```

Algorithm 4-5: Spiral algorithm to find the nearest neighbor to a point *x*

Example 4-2:

Two nearest neighbor queries are illustrated in Figure 4-6a. To find the nearest neighbor to point x_1 Z-region ρ_1 is retrieved. The two points y_1 and z_1 are the nearest neighbors to x_1 on ρ_1. Since no point on the border of ρ_1 has a distance less than the distance of y_1 or z_1, we already have found the nearest neighbors. To find the nearest neighbor to point x_2 we first retrieve Z-region ρ_3. On ρ_3 we find the nearest neighbor y_2. Since a point of the border of ρ_2 has a closer distance to x_2 than y_2, we have to retrieve ρ_2. Since no point of ρ_2 has a closer distance than y_2 and no further border of a region has a closer distance to x_2 than y_2, we have found the nearest neighbor.



(a) (b)

Figure 4-6: Three nearest neighbor queries

To find the nearest neighbor to point x in the UB-Tree of Figure 4-6b, initially the Z-region ρ_1 is retrieved, yielding the nearest neighbor y. Since points on the border of ρ_2 have a distance to x that is less than the distance of y, next the region ρ_2 is retrieved. ρ_2 contains the point z, which has the distance a_2 to x, which is less than the distance a_1 of y. Therefore the nearest neighbor candidate now is z. Since no points on the border of not retrieved Z-regions have a lower distance to x than z, the algorithm terminates with z as the nearest neighbor.

Implementation and further analysis of the spiral algorithm is performed by a master student supervised by the author [Str99]. Detailed analysis of the algorithm can be found there. We just stress that the algorithm in worst case results in a range query. The query box of that range query is determined by the covering square to a circle around x. The radius of the circle is defined by the distance of the nearest neighbor to x.

4.7 The Tetris Algorithm for Sorted Processing of Query Boxes

Tables organized by a UB-Tree can be read in any sort order in O(n) disk accesses where n is the number of pages of the table or the minimal number of regions covering a query box. [Bay97] proposes to partition the universe in a number of equally sized slices. Algorithm 4-4 is used to retrieve each slice. After retrieval a slice is sorted and returned to the caller of the sort operation.

For non-uniformly distributed universes we propose a modification of the range query algorithm and a caching technique, the so called ”Tetris-Algorithm”. The Tetris algorithm is a generalization of the multidimensional range query algorithm (cf. Section 4.4) that efficiently combines sort operations with the evaluation of multi-attribute restrictions. The basic idea is to use the partial sort order imposed by a multidimensional partitioning in order to process a table in some total sort order. Essentially a plane sweep [PS85] over a query space defined by restrictions on a multidimensionally partitioned table is performed. The direction of the sweep is determined by the sort attribute. Initially the algorithm calculates the first Z-region that is overlapped by the query box, retrieves it and caches it in main memory. Then it continues to read and cache the next Z-regions with respect to the requested sort order, until a complete thinnest possible slice of the query box (in the sorting dimension) has been read. Then the cached tuples of this slice are sorted in main memory, returned in sort order to the caller and removed from cache. The algorithm proceeds reading the next slice, until all Z-regions intersecting the query box have been processed. Only disk pages overlapping the query space are accessed. With sufficient, but modest, cache memory each disk page is accessed only once: To sort 50% of a 1.3 GB relation, the Tetris algorithm just requires a cache of 2.6 MB, whereas a standard merge-sort algorithm needs a cache of 750 MB (see sections 6.4.1 and 8.2.1).

```
Input:  $y,z$ : tuples defining a query box with $Z(y) \prec Z(z)$
        $i$   : sorting dimension
Output: stream of tuples in [[$y$, $z$]] sorted according to dimension
$i$

$\xi = Z(y)$; $\omega = Z(z)$
repeat
    search [$\alpha$:$\beta$] in the UB-Tree, such that $\alpha \preccurlyeq \xi \preccurlyeq \beta$
    store all tuples from page($\alpha$:$\beta$) in the cache
    if a new slice in dimension $i$ is completed
        $s$ = endpoint of the slice dimension $i$
        sort all cached tuples
        output all cached tuples $x$ where $x_i \le s$
        remove all cached tuples $x$ where $x_i \le s$ from cache
    end if
    $\xi$ = Z-address of the next point intersecting the querybox
    with respect to dimension $i$
until $\xi \succ \omega$
```

Algorithm 4-6: Tetris algorithm for a query box [[*y*, *z*]]

Thus the Tetris-Algorithm works similar to the range-query algorithm. The only difference is that the calculation of the next intersecting Z-region does not return a Z-region according to Z-ordering, but according to the specified sort order (Tetris order): Initially the algorithm calculates the first Z-region that is overlapped by the query-box, retrieves it and caches it in main memory. Then it continues to read and cache the next Z-regions with respect to the sort order, until a complete thinnest possible slice of the query box has been read. Then the cached tuples of this slice are sorted in main memory, returned in sort order to the caller and removed from cache. The algorithm proceeds reading the next slice, until all Z-regions which intersect the query box have been retrieved and output.

An example of sorted reading with the Tetris-Algorithm is illustrated in Figure 4-7, where the entire universe is sorted according to the vertical dimension, i.e., from bottom to top. The part of each region from which tuples are cached is shaded in this Figure. Note that a large volume of a cached region does not mean that a lot of cache memory is needed: The maximum number of tuples in each region and thus maximum cache size for each cached region is *C* tuples. The current sweep line is denoted by a thin white line. Thus the current slice processed by the Tetris algorithm is the shaded area up to the white line. The algorithm starts by retrieving the Z-region in the very left corner (Figure 4-7a). Successive Z-regions are retrieved and cached (Figure 4-7b) until the first vertical slice is completed (Figure 4-7c). The tuples of this slice are then sorted in main memory. All tuples up to the end coordinate of the slice in the sorting dimension are output in sort order and removed from cache. The Z-regions of this slice can not be removed from cache completely at this point since each Z-region still might have some tuples that have not been output yet. However, not the entire Z-regions are cached. Caching is only necessary for those tuples that have not been output yet. In Figure 4-7d one additional Z-region has been read to complete the second slice. Then eight more Z-regions are read and cached until the third slice is completed (Figure 4-7e). The Tetris-Algorithm continues processing in this way until the last slice of the query box has been read (Figure 4-7f), i.e., the complete universe has been read in the desired sort order.

The visualization also gives a hint why we named this algorithm *Tetris algorithm*: The caching order of the regions reminds us of the Tetris computer game. Regions are cached, until a slice in the sorting direction has been completed. Upon completion the slice is removed from cache.

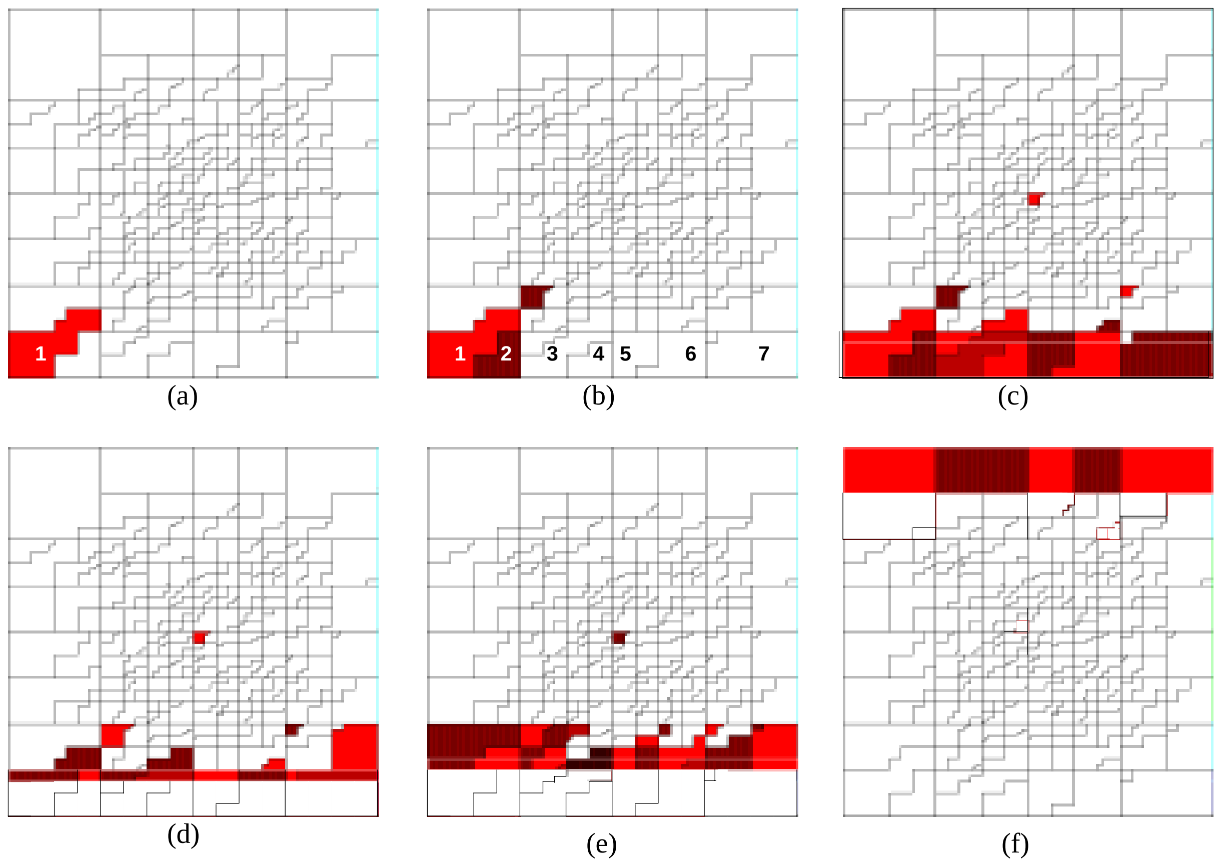


Figure 4-7: Sorted reading with the Tetris algorithm

Figure 4-8 shows the first three vertical slices that occur while the Tetris algorithm reads the thick bordered query box in sort order according to the vertical dimension. The cached Z-regions for each slice are shaded, the slices are emphasized by white borders.

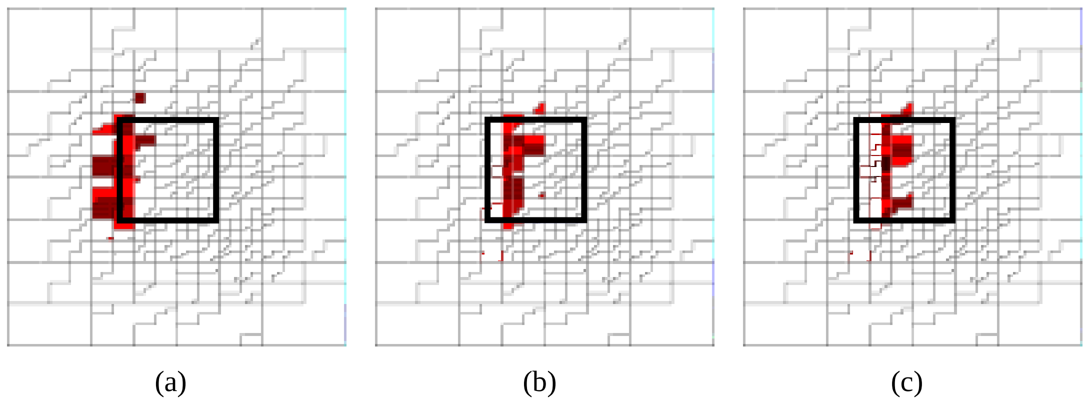


Figure 4-8: Reading a query box in sort order

The caching strategy of this algorithm can be improved even further. Figure 4-8 shows some Z-regions that might cache data outside the query box. Of course it is not necessary to cache this data allowing for an even smaller cache size.

Both the Tetris algorithm and the range query algorithm determine the set of Z-regions overlapping a query box. The only difference is the order in which the Z-regions are produced. While the range query algorithm delivers the Z-regions in Z-order, the Tetris algorithm uses an order which depends on the sort attribute A_j. We call this order Tetris order T_j [MZB99]. Formally, Tetris ordering produces a compound ordering from a Z-ordering by extracting the bits of the sort attribute from the Z-address and concatenating them to the remaining Z-address, i.e., for sorting with respect to attribute A_j the ordinal number $T_j(a)$ for a Z-address α is computed as:

$$T_j(a) = (Z^{-1}(\alpha))_j \circ Z((Z^{-1}(\alpha))_1, ..., Z^{-1}(\alpha))_{j-1,} Z^{-1}(\alpha))_{j-1} ,..., Z^{-1}(\alpha))_d)$$

The ordinal number $T_j(x)$ for Cartesian tuple x is computed respectively:

$$T_j(x) = x_j \circ Z(x_1, ..., x_{j-1},x_{j+1},...,x_d)$$

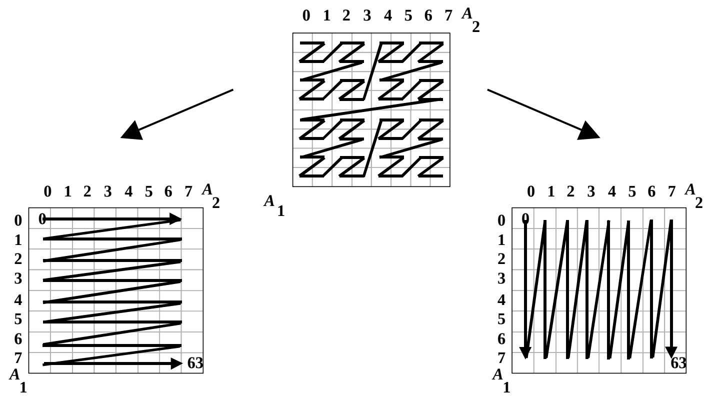


Figure 4-9: Z-Ordering / Tetris Ordering

Figure 4-9 shows the Tetris orderings T_1 and T_2 created from a Z-ordered 8×8 universe. Although Tetris ordering looks like a compound ordering in the Figure, this is only true for the two-dimensional case. In general, Tetris ordering is a concatenation of one attribute with the Z-address of a tuple reduced by one attribute.

Although the idea of the Tetris algorithm has been developed by the author, implementation and a detailed analysis are beyond the scope of this thesis. Some very basic performance considerations are given in Chapters 5 and 7. We give a more detailed description of the Tetris algorithm in [MZB99].

Errors, like straws, upon the surface flow; he who would search for pearls must dive below.

(John Dryden)

Chapter 5

Prototype Implementation

Relying on the built-in B-Tree of a DBMS, the UB-Tree is easily implemented on top of that DBMS. In this chapter we give an overview of the prototype implementation of UB-Trees. We describe basic implementation concepts and specifically illustrate the calculation of Z-addresses, since this calculation is crucial to both flexibility and performance of our algorithms. Next to the basic bit-interleaving algorithm, we illustrate how Z-addresses for arbitrary data types are calculated by the concept of transformation functions. Then we give examples for three transformation functions: For signed integer numbers it suffices to invert the sign bit to use bit-interleaving. To deal with independently distributed dimensions the concept of Variable UB-Trees (VUB-Trees) is introduced. Multidimensional hierarchical clustering (MHC) is useful for clustering a universe where hierarchies are built over the domain of each dimension. After the concept of transformation functions we address specific algorithmic problems that we had to solve when implementing the insertion, deletion, point- and range query algorithms. Finally we describe the functionality that is provided by the UB-Tree library, a C-library offering UB-Tree functionality.

5.1 Overview of the Prototype

To practically evaluate the performance of the UB-Tree, a basic C-library providing the algorithms described in this thesis was implemented by the author. Several master students and interns supervised by the author implemented applications on top of the C-library:

- *create* – a program to fill a database with generated test data of various data distributions, dimensionality, and size [Fri97]
- *rtest* – a benchmark program to measure the query performance of UB-Trees and various other indexes [Fri97]
- *regvis* – a program to visualize the region partitioning of UB-Trees [Fri97]
- *load* – a mass loading tool to build large databases quickly [Bau97]

Several master students and interns enhanced the C-library [Sch98, Bau98, Fen98] and helped to port the original TransBase/Solaris code to further RDBMS and operating systems [Pie97, Ova99, Pfa99].

The prototype implementation of the UB-Tree was performed on top of the RDBMS TransBase on Solaris. The current version of the UB-Tree library supports the RDBMSs TransBase, Oracle 8 [Pie97] and DB2 [Ova99], and the operating systems Solaris 2.6, Windows NT [Pie97], and Linux [Bau98]. In addition the library was ported to Informix Universal Server [Pfa99]. The complexity of the library can be seen from the module hierarchy of Figure 5-1.

The entire system is implemented in ANSI C and ESQL and relies on the GNU tools gcc and gmake for code generation. The Oracle version relies on the Oracle Call Interface (OCI, see [Ora97]) and the DB2 version uses CLI [IBM97] to communicate with the DBMS server. Automated documentation generation is enabled by using doc++ [WZ98]. cvs [CVS98] is used for version management.

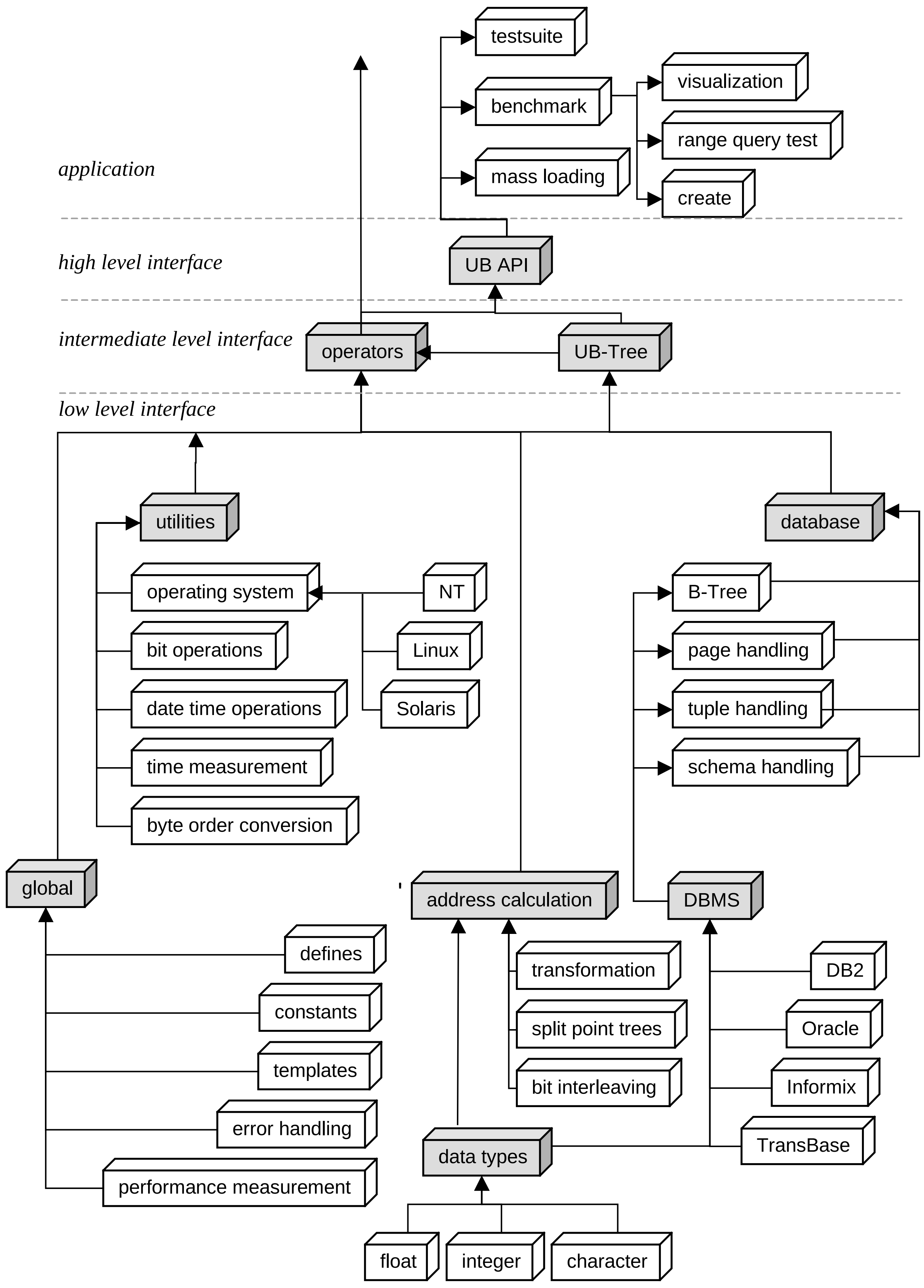


Figure 5-1: Architecture of the pilot implementation

5.2 Basic Implementation Concepts

The UB-Tree is realized on top of a relational DBMS by utilizing the B*-Tree of that RDBMS. To show the easy portability of the UB-Tree library, we ported the initial TransBase 5.4 implementation to Oracle 8, DB2 UDB V5, and Informix Universal Server.

Each UB-Tree is a logical construct, which is physically realized by a relational table. We call that relational table from now on *UB-Tree index table*. A UB-Tree index table contains one tuple for each page of the UB-Tree. As listed in Table 5-1, each tuple of the UB-Tree index table stores the Z-address of the corresponding Z-region, the number of tuples stored on the data page corresponding to that Z-region, and the content of the page corresponding to that Z-region. The primary key of this table is the Z-address.

Note that the tuples stored in a UB-Tree do not correspond to the tuples stored in a UB-Tree index table: the number of tuples in the UB-Tree index table rather is the number of data pages of the UB-Tree. In the remainder of this chapter a tuple of the UB-Tree index table is called *UB-page*, whereas *UB-tuple* denotes a tuple that is stored in the UB-Tree, i.e., a tuple that is stored on a UB-page.[13]

attribute	data type	description
Z-address	bit data	upper Z-address of the Z-region corresponding to that UB-page
number of tuples	number	number of tuples stored on that UB-page
data page content	bit data	content of the UB-page

Table 5-1: Schema of a UB-Tree index table

The DBA has to ensure that each UB-page (i.e., each tuple of the UB-Tree index table) is stored on a separate physical disk page. This is achieved by choosing the size of each UB-page (size of the attributes *Z-address, number of tuples, data page content* plus some DBMS storage overhead) to be exactly the size of a physical database page. For some DBMS additional settings have to be done to ensure the physical correspondence of DBMS pages and UB-pages [Pie97, Ova98]. The physical correspondence allows one to implement the UB-Tree on top of a RDBMS while obtaining the I/O-behavior identical to a kernel implementation of UB-Trees. However, significantly higher CPU- and inter-process communication cost and an impedance mismatch are the price that must be paid for this simple implementation approach:

[13] A straightforward idea for an implementation of the UB-Tree index table is to enhance each tuple of a relation with one additional column, namely its Z-address. The Z-address as primary key then leads to a multidimensional clustering [OM84]. However, this approach does not offer control over region boundaries during a page split; the region boundaries are defined by the tuples in this case. Our approach offers complete control over the choice of the region boundary during a Z-region split. We also preferred this approach since the UB-Tree is easier to integrate into a RDBMS kernel with this implementation.

- The UB-tuples on each UB-page must be extracted and set by special functions provided by the API of the UB-Tree library (UBAPI). The standard SQL functionality of the DBMS cannot be used to achieve that.

- Each query or update operation requires two interprocess communications and ESQL parsing operations for each UB-page that needs to be accessed by the operation.

- The performance of the prototype implementation heavily relies on the optimizer of the underlying DBMS, since SQL statements are used to access UB-pages.

The UB-Tree algorithms require an efficient access to the Z-addresses stored in the UB-Tree. Therefore, a secondary B-Tree on *Z-address* is created and builds the UB-index part of Figure 4-1. The UB-Tree index table builds the UB-file illustrated in that figure.

Table 5-2 lists the prototype implementation concepts and compares their implementation in TransBase 4.3, Oracle 8, and a DBMS kernel implementation.

UB-Tree concept	TransBase 4.3 implementation	Oracle 8 implementation	DBMS Kernel Implementation
UB-Tree index table	relation	relation	clustering primary index
UB-page	tuple with the size of a physical page	tuple with the size of a physical page	physical DBMS page
UB-tuple	substring of the data page content attribute of the UB-Tree index table	substring of the data page content attribute of the UB-Tree index table	DBMS tuple
Z-region	defined by the Z-address attribute of the UB-Tree index table	defined by the Z-address attribute of the UB-Tree index table	binary string
UB-index	additional table storing the Z-address attribute of the UB-Tree index table	secondary index on the Z-address attribute of the UB-Tree index table	B-Tree node pages
UB-file	relation storing the UB-Tree index table	relation storing the UB-Tree index table	B-Tree leaf pages

Table 5-2: Prototype implementation concepts

In TransBase it is necessary to use an IOT on the *Z-address* to implement the UB-Tree index table, since TransBase always clusters the data according to the primary key. Since UB-pages are physical DBMS pages, TransBase needs to perform a random leaf page access on the IOT to retrieve a Z-address. An efficient retrieval of a range of Z-addresses is not possible via the UB-Tree index table, since it is not possible to create a secondary index on a primary key attribute. Therefore the TransBase implementation introduces a new concept, a *UB-Tree secondary index table*, which stores only the Z-addresses of the UB-Tree index table. This allows an efficient access to a range of Z-addresses: While only one Z-address is stored on a leaf page of the UB-Tree index table, a set of Z-addresses in consecutive order is stored on a leaf page of a UB-Tree secondary index table. This greatly reduces the number of random page accesses and significantly improves the range query and Tetris performance of the prototype implementation on TransBase. However, a larger cache requirement is the price for this implementation. The upper nodes of both the UB-Tree index table and the UB-Tree secondary index table need to be cached in main memory (see Figure 5-2a). In contrast to that the Oracle implementation just caches the UB-Tree index table (see Figure 5-2b). This is the reason for performance gain of 12% of our prototype implementation on Oracle for tables larger than one 1GB [Pie98].

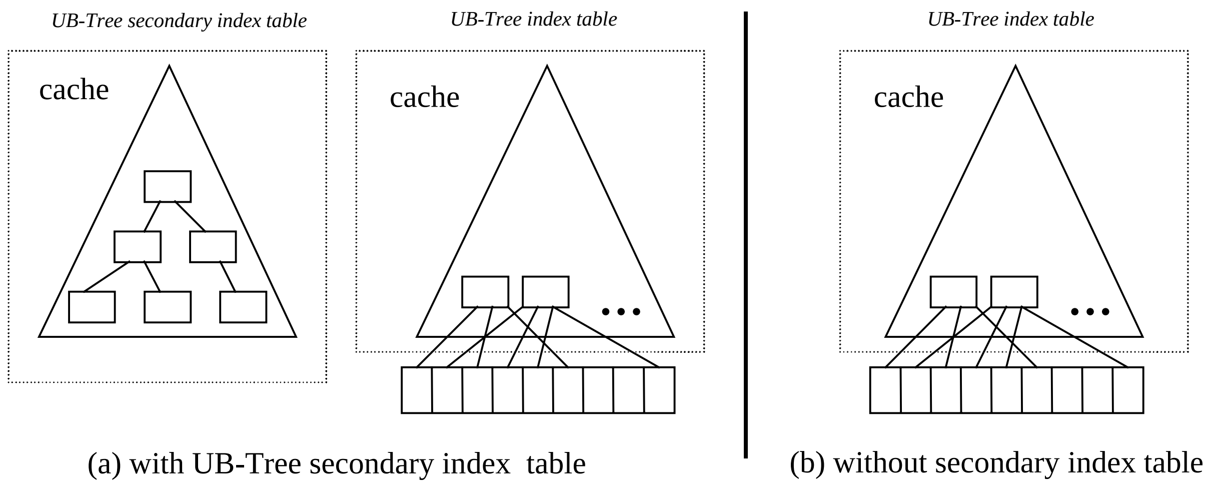

UB-Tree secondary index table
cache
UB-Tree index table
cache
UB-Tree index table
cache
(a) with UB-Tree secondary index table
(b) without secondary index table

Figure 5-2: Cache requirements and UB-Tree secondary index table

In the remainder of this chapter we illustrate selected algorithmic problems and performance problems which we had to solve in order to realize our prototype.

5.3 Implementing the Address Calculation

The address calculation algorithm first relied solely on bit-interleaving and supported only positive integer numbers. Later the architecture was enhanced to allow plug-in functions for any arbitrary data type. Since certain data distributions cause a multidimensional partitioning with a high puff pastry degree and thus a bad performance of the UB-Tree algorithms, the address calculation was enhanced to take any data distribution into account. Another enhancement envisioned by the author is to modify bit interleaving to weigh attributes for the interleaving process.

5.3.1 Bit-Interleaving

The performance of the UB-Tree crucially relies on an efficient implementation of the Z-address calculation. For tuples of positive integer numbers Definition 3-3 immediately yields an algorithm to calculate the standard address from the binary representation of a tuple: Z-values can be calculated by interleaving the bits of the attributes in a certain order.

In practical applications one often wants to index a multidimensional domain, where the cardinality is not identical for all one-dimensional domains. Only a slight modification of the interleave operation is necessary to support a universe where the domain of each dimension does not consist of the same number of bits steps(r), i.e., there is some $i, j \in D$ so that $r_i \neq r_j$. In this case the number of bits is not identical for each step of an address, but step k consists of steplength(k) bits. Algorithm 5-1 shows this generalized algorithm for bit-interleaving.

```
Input:  x : tuple
Output: Z-address $\alpha$

// the algorithm requires the dimensions to be
// sorted according to their resolution
// in descending order

for step = 1 to max({steps($r_j$) | $j \in$ D})
        for i = 1 to steplength(step)
            copy bit step of $x_i$ to bit i of $\alpha_{step}$
        end for
end for
```

Algorithm 5-1: Bit-interleaving to calculate $\alpha = Z(x)$

With r = max({steps(r_j) | $j \in$ D}) Algorithm 5-1 has a CPU-complexity of $O(d \cdot r)$ bit operations (resp. $O\left(\sum_{i=1}^{d} r_i\right)$ for attributes of different length). The same holds for Algorithm 5-2 to calculate the Cartesian coordinates of a tuple from its address.

```
Input:  $\alpha$ : Z-address
Output: x : tuple

// the algorithm requires the dimensions to be
// sorted according to their resolution
// in descending order

for step = 1 to max({steps($r_j$) | $j \in D$})
    for i = 1 to steplength(step)
        copy bit i of $\alpha_{step}$ to bit step of $x_i$
    end for
end for
```

Algorithm 5-2: Bit extraction to calculate $x = Z^{-1}(\alpha)$

Algorithm 5-3 to extract one specific Cartesian co-ordinate from a Z-address has the CPU-complexity $O(r_i)$.

```
Input:  α : Z-address
        i : number of attribute to extract
Output: x_i : attribute i of tuple x

// the algorithm requires the dimensions to be
// sorted according to their resolution
// in descending order

for step = 1 to r_j
    copy bit i of α_step to bit step of x_i
end for
```

Algorithm 5-3: Bit-extraction to calculate $x_i = (Z^{-1}(\alpha))_i$

As one can see from the above algorithms, the calculation of Z(*x*) and $Z^{-1}(x)$ can be implemented efficiently by simple bit operations. The linear behavior of the bit interleaving performance for tuples consisting of 32 bit integers for 2 up to 10 dimensions on a SUN ULTRA SPARC 143 MHz is displayed in Figure 5-3. The Figure shows that Z-address calculation takes less than 1ms even for 10-dimennsional Z-addresses. Thus Z-address calculation is more than one order of magnitude faster than a random disk access, which usually takes 10ms. Using state-of-the-art CPUs with 300 MHz and more, the Z-address calculation is more than 100 times faster than a random disk access and thus is negligible.

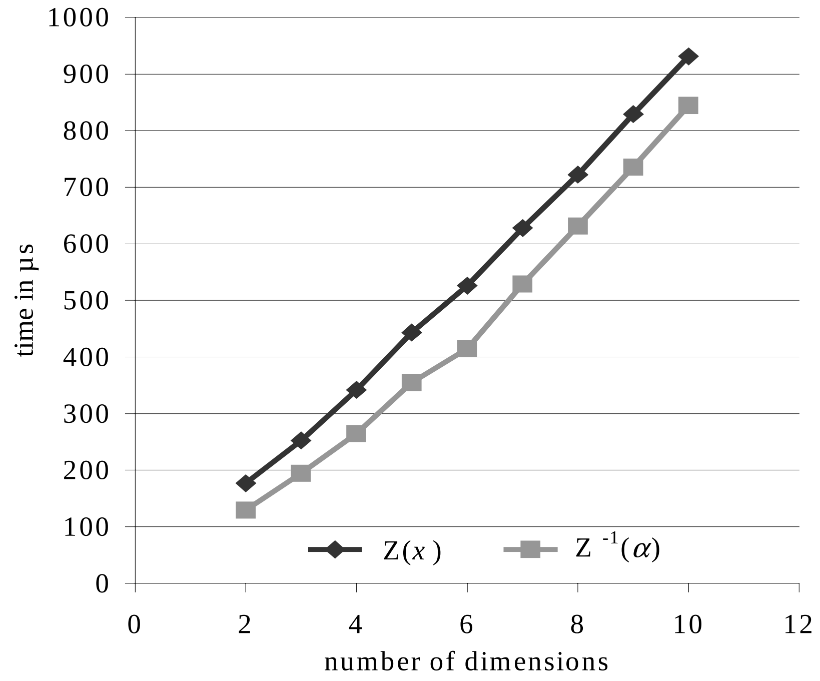


Figure 5-3: CPU time to calculate Z(*x*) and $Z^{-1}(\alpha)$

5.3.2 Address Calculation for Arbitrary Data Types

We described bit-interleaving for address calculation just for positive integer numbers. Algorithm 5-1 can be used to calculate the Z-address for any data type, when the bit-lexicographic order on the binary representation of tuples matches the “natural” ordering on this type.

Definition 5-1 (lexicographic character order $<_{C1\circ Cn}$, lexicographic byte order $<_{B1\circ Bn}$, lexicographic bit order $<_{b1\circ\,\ldots\,\circ bn}$): For the following examples we write $<_{C1\circ Cn}$ for the lexicographic order on the character concatenation $C_1\circ\ldots\circ C_n$, $<_{B1\circ Bn}$ for the lexicographic order on the byte concatenation $B_1\circ\ldots\circ B_n$ and $<_{b1\circ\,\ldots\,\circ bn}$ for the lexicographic order on the bit concatenation $b_1\circ\ldots\circ b_n$.

Lemma 5-1: For positive integer numbers with a fixed length binary representation the bit lexicographic order on the binary representation is identical to the <-order on integer numbers.

Proof:
The proof is a direct consequence of the definition of binary numbers and of the definition of lexicographic order. □

Example 5-1:

$65_{10} < 66_{10}$ maps to the binary representation
$01000001_2 <_{b1\circ\,\ldots\,\circ b8} 01000010_2$

Lemma 5-2: For character strings the bit lexicographic order on the binary representation is identical to the <-order on integer numbers.

Proof:
The proof is a direct consequence of Lemma 5-1, the definition of ASCII codes and the fact, that character strings are a concatenation of characters, which are ordered lexicographically. □

Example 5-2:

“AB” $<_{C1\circ Cn}$ “BB”
maps to the ASCII-Codes
$65_{10} \circ 66_{10} <_{B1\circ B2} 66_{10} \circ 66_{10}$
maps to the binary representation
$01000001_2 \circ 01000010_2 <_{b1\circ\,\ldots\,\circ b16} 01000010_2 \circ 01000010_2$

Because of Lemma 5-1 and Lemma 5-2 bit interleaving can be used to calculate the addresses for character strings and positive integer numbers.

If for a data type the bit-lexicographic order on the binary representation of tuples does not match the “natural” ordering on this data type, a transformation function with that desired property must be defined. For address calculation this transformation function is applied on the attribute prior to bit interleaving. Transformation is bijective. For Cartesian co-ordinates the inverse function to the transformation function is applied after bit extraction. Figure 5-4 shows the architecture for the address calculation for arbitrary data types.

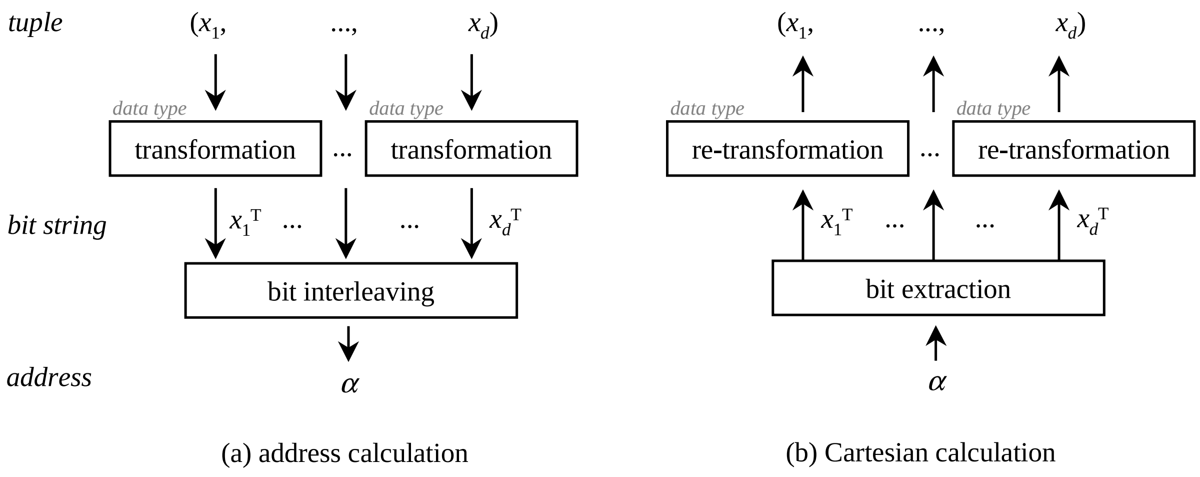


Figure 5-4: Dealing with arbitrary data types

It is out of the scope of this thesis to analyze transformation functions for further data types. We just give a definition of transformation functions here:

Definition 5-2 (transformation function): We call a function f from any ordered domain ($\mathbb{D}$, <) to a binary string a *transformation function* , if and only if, for a, $b \in \mathbb{D}$:

$$a < b \Leftrightarrow f(a) <_{b1\circ \dots \circ bn} f(b)$$

For positive integers and character strings the transformation function is the identity function. For integer numbers in complement representation the simplest transformation function is to invert the sign bit. Transformation functions for real numbers and date/time data types were implemented under the supervision of the author and are described in the MISTRAL documentation [MOD99]. Note that in order to do any algorithm from the previous chapter like the Tetris algorithm, the spiral algorithm, or the range query algorithm we do neither require a transformation function to be bijective nor injective. However, if a transformation function is bijective, re-transformation to derive Cartesian tuples from an address is possible.[14]

5.3.3 Dealing with low Cardinality Domains: Enumeration Types

Character strings very often have a similar prefix in the first bits. All capital letters of the roman alphabet in ASCII have an identical binary prefix 010_2, all numbers have a constant

[14] Note that because of their binary nature, transformation functions are hardware dependent. One specific problem of our implementation was to deal with little/big endian representation [PH90] of machine words and binary strings in various microprocessors.

binary prefix of 0011_2. Thus combining character strings and integer numbers easily results in a strong puff-pastry of the UB-Tree.

Many applications use the domain of character strings merely to represent an enumeration type, i.e., a domain consisting of a small set of distinct values. A typical example is the SHIPMODE attribute of the LINEITEM table of the TPC-D benchmark. SHIPMODE has the domain of character strings, although the set {REG AIR, AIR, RAIL, SHIP, TRUCK, MAIL, FOB} represents all permissible values for SHIPMODE. There is no order on the values of SHIPDMODE. For instance, it makes no sense to ask for all tuples satisfying the textual comparison SHIPMODE < “AIR”.

Definition 5-3 (enumeration type): We call the data type of an attribute to be an *enumeration type*, if its actual domain $\mathbb{V} \subseteq \mathbb{D}$ consists of a relatively small finite set of values, usually listed explicitly.

In order to maximize the entropy of an enumeration type of a domain $\mathbb{D}$ we define an order preserving one-to-one map f and its inverse function f^{-1}:

$$f : \mathbb{D} \rightarrow \{0, 1, ..., |\mathbb{D}| - 1\}, \text{ such that for } a, b \in \mathbb{D}\text{: } f(a) <_{\mathbb{N}} f(b) \Leftrightarrow a <_{\mathbb{D}} b$$

If there is no reasonable ordering on an enumeration type, we drop the requirement on f to be order preserving and merely require:

$$f : \mathbb{D} \rightarrow \{0, 1, ..., |\mathbb{D}| - 1\}, f \text{ injective}$$

We call f a *surrogate function* for an enumeration type. For each value $a \in \mathbb{D}$ we call $f(a)$ the *surrogate* of a. For a very compact representation we number surrogates in sequential order.

$0 = 000_2 = f(\text{REG AIR})$	$0 = 000_2 = g(\text{REG AIR})$
$1 = 001_2 = f(\text{AIR})$	$4 = 100_2 = g(\text{AIR})$
$2 = 010_2 = f(\text{RAIL})$	$2 = 010_2 = g(\text{RAIL})$
$3 = 011_2 = f(\text{SHIP})$	$6 = 110_2 = g(\text{SHIP})$
$4 = 100_2 = f(\text{TRUCK})$	$3 = 011_2 = g(\text{TRUCK})$
$5 = 101_2 = f(\text{MAIL})$	$7 = 111_2 = g(\text{MAIL})$
$6 = 110_2 = f(\text{FOB})$	$5 = 101_2 = g(\text{FOB})$

Table 5-3: Two surrogate mappings for an enumeration type

The surrogate mapping f of Table 5-3 for the enumeration type SHIPMODE uses running numbers from 0 to 6. The function g defines a mapping which is more suitable than the mapping f with respect to the space partitioning of UB-Trees: If the order of values from top to bottom in Table 5-3 represents the insertion order of SHIPMODE values, even for an actual domain consisting only of the first four values a puff pastry as discussed in Section 3.10 is avoided. This is not true for the mapping f since the first four values have the leading bit as common prefix. Thus if an enumeration type is built on the fly (i.e., its actual domain may grow during the course of time), a surrogate function should vary the leading bits first, since these bits will be used for space partitioning. Using running numbers for the surrogate of an enumeration type may result in a sub-optimal space partitioning, since the leading bit might not be used (see Section 3.10). If running numbers are used and the binary representation of each surrogate is reverted with the function mirror($b_1...b_n$) = $b_n...b_1$, the leading bit of the surrogate will already partition the domain. However, since the natural order on integers is lost by mirroring, this mapping should not be used, if an enumeration type is not only restricted to points, but also to ranges.

To store an enumeration type in a UB-Tree, we map the enumerated set bijectively onto a set of surrogates. For v distinct values in the enumeration, only $\lceil \log_2 v \rceil$ bits are used to represent the enumerated set. Since all of these bits are used, a puff pastry is avoided to a large extent. Thus the number of Z-regions overlapping the query box but not contributing to the result set is minimized by that mapping.

Note that if the values are not uniformly distributed, a puff pastry effect will still exist. However, the degree of the puff pastry will be much lower than for the character string representation. If an enumeration is not required to be order preserving, mirroring the bits of a surrogate may help to further reduce probability of a puff pastry.

5.3.4 Multidimensional Hierarchical Clustering

Often, especially in data warehousing applications, hierarchies are built on enumeration types to provide structure to otherwise flat dimensions. Hierarchies then are used to provide an appropriate method of describing the levels of semantically meaningful aggregations for a dimension. With a relational modeling this means that some attributes of a table are functionally dependent. We call this special kind of functional dependence *hierarchical dependence*.

Definition 5-4 (hierarchical dependence): We call an ordered sequence of n attributes $A_1 \circ A_2 \circ A_3 \circ ... \circ A_{n-1} \circ A_n$ to be *hierarchically dependent,* if and only if for any $i \in \{1, ..., n\}$ the domain of attribute A_i functionally depends on the values of attributes $A_1,...,A_{i-1}$

Definition 5-5 (hierarchical independence): We call a sequence of attributes $A_1 \mid A_2 \mid A_3 \mid ... \mid A_{n-1} \mid A_n$ to be *hierarchically independent,* if they are not hierarchically dependent.

OLAP queries often impose restrictions with respect to hierarchies over multiple dimensions. These restrictions usually are point restrictions or interval restrictions on some hierarchy level [Sar97]. The result set satisfying these restrictions is usually quite large; for presentation it is grouped and aggregated or ranked. Clustering data with respect to multiple hierarchies can substantially speed up these operations.

5.3.4.1 Hierarchies on Dimensions

For our definition of multidimensional hierarchical clustering we use a set concept to formally define hierarchies: A dimension *A* consists of a base type having a set of values $\mathbb{D}$ ={v_1,...,v_n}. A *hierarchy* of *depth h* over *A* is an ordered set of levels, i.e., $H=\{\mathbb{L}_0 ,..., \mathbb{L}_h\}$ (see Figure 5-5). Each *hierarchy level i* of *H* over *A* is a set of sets $\mathbb{L}_i = \{ m_1^i, ..., m_j^i \}$ with $m_k^i \subseteq \mathbb{D}$ for *k*=1,..,*j*. Each $m \in \mathbb{L}_i$ is a member set (or member) of the hierarchy of level *i* containing all members of a category. Usually a member *m* is assigned a name *label*(*m*) (e.g., 'Orange Juice' for m_1^1) instead of enumerating all values $v_k \in m$. The subset relationship $\subseteq$ between the members of two neighboring levels $\mathbb{L}_i$ and $\mathbb{L}_{i+1}$ defines a hierarchical relation (i.e., partial ordering) between the levels (e.g., the product 'OJ0,7L' is in the product category 'Orange Juice'). Increasing the level of a hierarchy increases the *granularity* of the categorization, i.e., the data is classified according to finer categories.

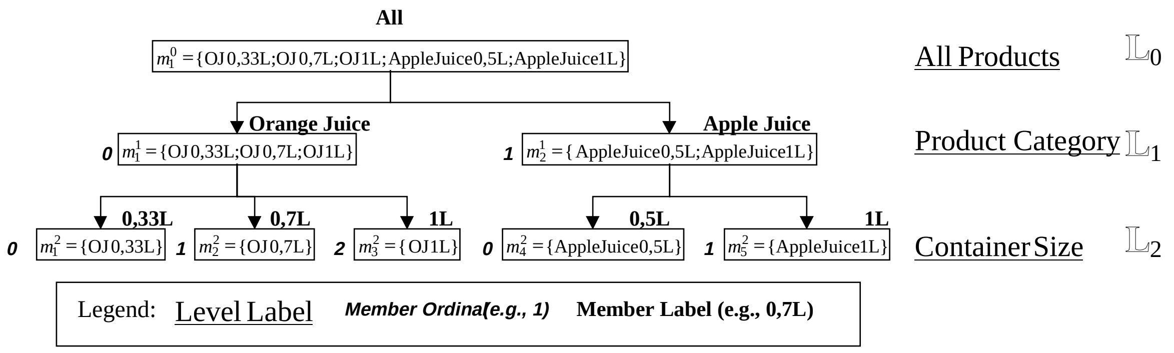


Figure 5-5: Example hierarchy in member set representation

The nodes of *H* are the hierarchy members (or member labels) connected by edges which are defined by the subset relationship between members of neighboring levels. The *children* of a member m_k^i of level *i* are all members m_l^{i+1} of the lower level *i*+1 that are subsets of m_k^i, i.e., children(m_k^i)={ $m_l^{i+1} \in \mathbb{L}_{i+1} \mid m_l^{i+1} \subseteq m_k^i$ } (e.g., the set {{'Apple Juice 0,5L'},{ 'Apple Juice 1L'}} is the children set of 'Apple Juice').

With

(1) the base set $\mathbb{D}$ as the only member of level $\mathbb{L}_0$ (i.e., $\mathbb{L}_0 = \{\mathbb{D}\}$)

(2) $m_k^i \cap m_l^i = \emptyset$ for all *i*, *k*, *l* and $k \neq l$

(3) $\bigcup \text{children}(m_k^i) = m_k^i$ for all *i*, *k*

a hierarchy *H* builds a hierarchy tree[15] with the root level $\mathbb{L}_0$. The *parent* of a member m_k^i of level *i* then is the member m_l^{i-1} of the upper level *i*-1 that is a superset of m_k^i, i.e., *parent*(m_k^i)={ $m_l^{i-1} \in \mathbb{L}_{i-1} \mid m_l^{i-1} \supseteq m_k^i$ } (e.g. 'Orange Juice' is the parent of 'OJ 0,7L').

[15] We will explain how to deal with complex hierarchies (i.e., directed acyclic graphs) in Section 5.3.4.4. Formally these hierarchies are modeled by dropping the requirement on *H* to be an ordered set of levels. Neighboring levels are then defined by coarsest refinement.

The bijective function ord_m defines a numbering scheme for the children of a member *m* of *H*. Ord_m assigns each subset (child) of *m* a number between 0 and the total number of children of *m* i.e.,

$$ord_m : children(m) \rightarrow \{0,\ldots,|children(m)|-1\}$$

(see Figure 5-5 for an example).

Hierarchies should never relate members of different dimensions, since dimensions are independent and thus such a hierarchy could be split up in two separate hierarchies (see Section 5.3.4.4).

5.3.4.2 Clustering Hierarchies

For each dimension hierarchy OLAP queries usually restrict some hierarchy level to a point. Sometimes an ordering on the levels of a hierarchy exists (e.g., a time hierarchy has an order on the days, months and years). Then interval restrictions on a hierarchy level may also occur. Our goal is to cluster data with respect to that partial order defined by the hierarchy levels.

For queries with large result sets one-dimensional clustering reduces disk accesses by a factor of *P*. Clustering of one-dimensional objects and single object hierarchies has been discussed to a large extent (e.g., [ZSL98], [BK89], [Sal88]).

If the order of dimensions during drill down is known in advance, clustering the data in this order will result in a good query performance. In principle, a concatenated clustering index (i.e., B^*-Tree) on the hierarchy levels of all dimensions in *one* lexicographic order is maintained. However, with *d* dimensions with h_i hierarchy levels over dimension *i*, there are $(\Sigma_{i=1}^{d} h_i)! / \Pi_{i=1}^{d}(h_i!)$ possible lexicographic orderings. For a 4-dimensional data cube (with 4 hierarchy levels for dimension 1, 4 for dimension 2, 2 for dimension 3 and 2 for dimension 4) there are 207900 possible orderings. Thus there is a high probability that the pre-defined clustering order will not be very useful for a particular query.

5.3.4.3 Encoding Hierarchies by Surrogates

To efficiently encode hierarchies, we introduce the concept of *compound surrogates* for hierarchies. Since we require hierarchies to form a disjoint partitioning, a uniquely identifying compound surrogate for each child node of a hierarchy member exists and can be recursively calculated by concatenating (∘) the compound surrogate of the member with the running number of the child node as calculated by the surrogate function *ord* from Section 5.3.4.1.

Definition 5-6 (compound surrogate): For a member m^i of hierarchy level *i* of hierarchy *H* we define its compound surrogate:

$$cs(H,m^i) = \begin{cases} ord_{father(m^i)}(m^i) & \text{, if } i = 1 \\ cs(H, \text{father}(m^i)) \circ ord_{father(m^i)}(m^i) & \text{, otherwise} \end{cases}$$

Definition 5-7 (path through a hierarchy): A path Φ through a hierarchy of depth h is specified by a list of members $m^1, ..., m^h$, where m^i is a member of hierarchy level i.

The compound surrogate for a hierarchy path Φ then is calculated as:

$$cs(H,\Phi) = \mathrm{cs}(H, m^h) = ord_{father(m^1)}(m^1) \circ ord_{father(m^2)}(m^2) \circ ... \circ ord_{father(m^h)}(m^h)$$

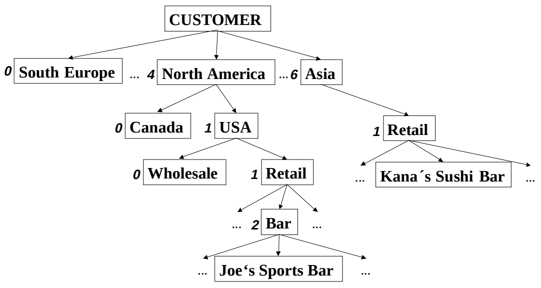


Figure 5-6: Part of a hierarchy

The hierarchy path North America → USA → Retail → Bar (Figure 5-6) has the compound surrogate:

$$ord_{\text{Customer}}(\text{North America}) \circ ord_{\text{North America}}(\text{USA}) \circ ord_{\text{USA}}(\text{Retail}) \circ ord_{\text{Retail}}(\text{Bar}) = 4 \circ 1 \circ 1 \circ 2$$

The upper limit of the domain for surrogates of level i is calculated as the maximum fan-out (number of children) minus one of all members of level i–1 of a hierarchy H, i.e.,

$$\text{surrogates}(H, i) = \max \{\text{cardinality}(\text{children}(H, m)) \text{ where } m \in \text{level}(H, i - 1)\} - 1$$

With $l_i = \lceil \log_2 \text{surrogates}(H, i) \rceil$ a *fixed length compound surrogate* can be stored in a very compact way by binary encoding.[16]

$$cs(H,\Phi) = \mathrm{cs}(H, m^h) = ord_{father(m^1)}(m^1) + ord_{father(m^2)}(m^2) \cdot 2^{l_1} + ... + ord_{father(m^h)}(m^h) \cdot 2^{l_1+l_2+...+l_{n-1}}$$

With n = 4 and l_1 = 3, l_2 = 1, l_3 = 3 and l_4 = 3 the above formula leads to the compound surrogate $cs(H, \text{Bar}) = 1000011010_2 = 538$.

Usually growth expectations for a hierarchy are known well in advance. Often hierarchy trees are even static. Therefore it is possible to determine a reasonable number of bits for storing each surrogate of the compound surrogate of a hierarchy. Since hierarchy trees grow exponentially, the overall number of bits necessary to store a compound surrogate is relatively small. For instance, a hierarchy tree with four branches on 8 levels already represents 4^8 = 65536 partitions and is stored by 16 bits.

[16] In general we use *variable length compound surrogates* that need $l_i(m) = \lceil \log_2 |\text{children}(m)| \rceil$ bits to store the surrogate for any child of m.

The lexicographic order on the hierarchy levels is preserved by this very compact fixed length encoding. Point restrictions on upper hierarchy levels result in range restrictions on the finest granularity of a hierarchy. For instance, the point restriction NATION = "USA" on the second level of the CUSTOMER hierarchy with *f*("North America") = 4 = 100_2 and *f*("USA") = 1 = 001_2 maps to the range restriction cs_{customer} between 528 = 1000010000_2 and 543 = 1000011111_2. Thus, a star join with this surrogate encoding for the foreign keys of a fact table results in a range restriction on each compound surrogate, if some hierarchy level of each dimension is restricted to a point (e.g., customer region = "USA"). In the same way intervals on the children of one hierarchy level result in a range of the corresponding compound surrogates (e.g., year = 1998 and month between April and June). A star join on a schema with *d* dimensions creates a *d*-dimensional interval restriction on the fact table.

5.3.4.4 Dealing with Complex Hierarchy Graphs

If two levels of a hierarchy graph are linked by several paths, there are several possibilities to define a hierarchy tree and therefore several ways to calculate the compound surrogates for physical clustering:

- If the order on the lowest level of granularity is identical for two hierarchy paths, then one path can be derived from the other path by an order preserving function on the lowest level of granularity. Then the clustering order for both hierarchy paths is identical. Thus, the clustering order for WEEK and MONTH in Figure 5-7a is identical. Both can be computed by an order preserving function from DAY, the lowest granularity level of the TIME hierarchy.
- If the query profile is known, the most useful path of the hierarchy graph used for restrictions, sort operations, or grouping should be chosen. Thus, if in Figure 5-7b queries on CUSTOMER usually restrict REGION and NATION, this path should be chosen for clustering.
- If the query profile is not known, all paths of a hierarchy graph may be used for clustering, since hierarchies may be used for restrictions independently during drill-down. For clustering, the different paths then can be considered to be independent dimensions. In the hierarchy graph of Figure 5-7b both the REGION hierarchy and the CUSTOMER hierarchy might be used for clustering. However, this approach increases the clustering dimensionality and thus should be used with care.

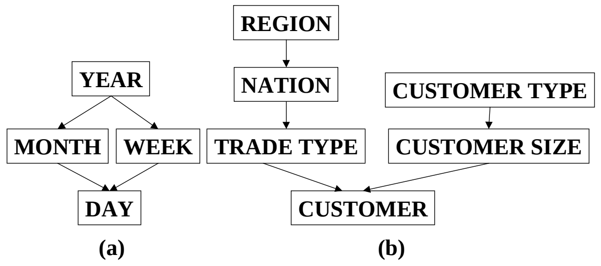


Figure 5-7: Complex hierarchy graphs

Other issues in the context of complex hierarchies are unbalanced hierarchies, slowly changing dimensions and multiple inheritance. Unbalanced hierarchies occur, if some hierarchy members have more child levels than others. This means, that the compound surrogates of Joe’s Sports Bar and Kana’s Sushi Bar in Figure 5-6 have different lengths. Using variable length compound surrogates or padding the shorter compound surrogate with zero bits solves this problem without any impact on clustering.

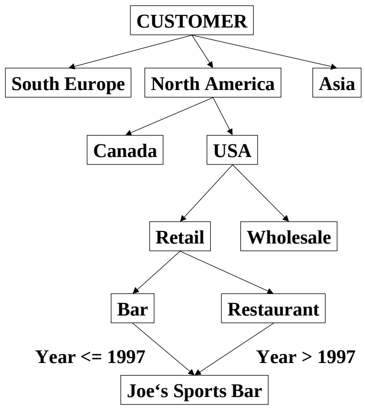


Figure 5-8: Slowly changing dimensions

Slowly changing dimensions can be addressed by marking each node of a hierarchy tree with a validity time interval. An object is physically clustered and retrieved with respect to its validity time. Re-organization of the physical clustering is not necessary: Even with a new classification upon a certain point of time the existing clustering should be correct from a historic perspective. If the business type of Joe’s Sports Bar changes from bar to restaurant in 1998 (cf. Figure 5-8), all previously clustered data still is correct. The total sales over all bars in 1997 must include Joe’s Sports Bar, whereas it is included in restaurants for 1998. However, each object of a hierarchy needs information about re-classification in order to correctly calculate the total sales to Joe’s Sports Bar over the last years.

Multiple inheritance (e.g., Joe’s Sports Bar is considered to be both a bar and a restaurant at the same time) is solved similarly to slowly changing dimensions: One of the several possible paths to a hierarchy node is chosen for clustering. The other paths of a hierarchy graph to that object then merely store a pointer to the sub-tree that actually stores the object. If multiple aggregation paths are possible, precautions must be taken that only one of these paths is used for aggregation.

5.3.4.5 Addressing Sparsity

Sparsity is defined as the percentage of a domain that is not existent in the actual domain. In data warehousing applications the multidimensional universe is often called *data cube*. For a multidimensional universe, i.e., data cube, sparsity is the ratio between the number of cells not containing any data and the overall number of cells of a data cube. Some OLAP tools allows one to mark dimensions to be sparsely populated and then specially handle them.

However, a multidimensional cube is formed as the cross product over the domains of all dimensions. Therefore, even for non-sparse dimensions the sparsity of the entire cube becomes extremely high soon. The ‘Juice & More’ schema of Section 8.3, for instance, is a star schema with four independent dimensions with a sparsity of 99,8%:

$$\text{sparsity(Juice \& More)} = 1 - \frac{26\text{ Mio}}{7030 \cdot 5600 \cdot 36 \cdot 12} = 0{,}9984 \text{ with sparsity(star schema)} = 1 - \frac{|\text{Fact Table}|}{\prod_{i=1}^{d} |\text{Dim Table } i|}$$

To our knowledge sparsities of more than 99% are typical for data warehousing applications. The TPC-D schema (see Figure 2-2), for instance, can be regarded to be a snowflake schema with shared hierarchies consisting of three independent dimensions:

- part + supplier (combined dimension with 0.8 million records coming from 0.2 million parts from 10 thousand suppliers)
- customer + order (combined dimension with 1.5 million orders from 150 thousand customers)
- time (2557 records for seven years on the aggregation level of a single day)

For a fact table of 6 million records (a TPC-D scaling factor of 1) the resulting data cube has a sparsity of more than 99,99999%.

Thus, in practice sparsity forbids to materialize an entire data cube of raw data. Physical data organization in a multidimensional array is only feasible for highly aggregated data. However, serious decision support applications require a deep drill down into interesting areas of a data cube. Therefore it is necessary to have a physical representation of a sparsely populated data cube that allows efficient access to some part of that cube. With multidimensional hierarchical clustering, drill down defines a subspace of a data cube by range restrictions in several dimensions. Therefore a method to cluster sparse data with respect to several dimensions in combination with an efficient range query and sort algorithm are necessary for efficient handling of drill down queries.

The surrogate calculating function of Section 5.3.4.3 can use any multidimensional access method to implement multidimensional hierarchical clustering. However, using any variant of R-Trees [Gut84, BKS+90, BKK96] may result in a sub-optimal performance, since R-Trees may subdivide the universe into overlapping tiles, which may result in multiple accesses to one disk page. Therefore the most interesting candidates are Grid-Files [NHS84], hB-Trees [LS90], or space filling curves in combination with one-dimensional access methods [OM84, Jag90]. It is very well suitable for the UB-Tree, since the UB-Tree hierarchically organizes the data space. The hierarchy is directly reflected by the binary representation of the compound surrogates. Thus bit-interleaving as used by the UB-Tree causes a multidimensional partitioning whose boundaries directly reflect the hierarchy levels. Partial match queries with point restrictions on upper hierarchy levels then result in multidimensional range queries.

5.3.4.6 Processing OLAP Queries on Multidimensionally Clustered Data with the Tetris-Algorithm

Figure 5-9 illustrates how the Tetris algorithm is used to calculate the total sales for each different fruit juice for all customers in Asia on relation partitioned by a two-dimensional UB-Tree and hierarchies on the CUSTOMER and PRODUCT dimensions. The restriction of the REGION to 'Asia' results in an interval in the CUSTOMER dimension. The same holds for the restriction to 'Juice' for PRODUCT. The boundaries of each query interval correspond to Z-region boundaries and thus minimize the number of Z-regions only partly contained in the query box. The query box is read in sort order from bottom to top; the aggregates for each juice type are calculated on the fly. The part of each Z-region from which tuples are cached is shaded. When all Z-regions intersecting the 'Orange Juice' slice have been read, this slice is sorted and aggregated. In the same way the next slices ('Apple Juice', 'Cherry Juice', etc.) are processed. This continues until the entire product interval defined by the restriction to 'Juice' has been handled.

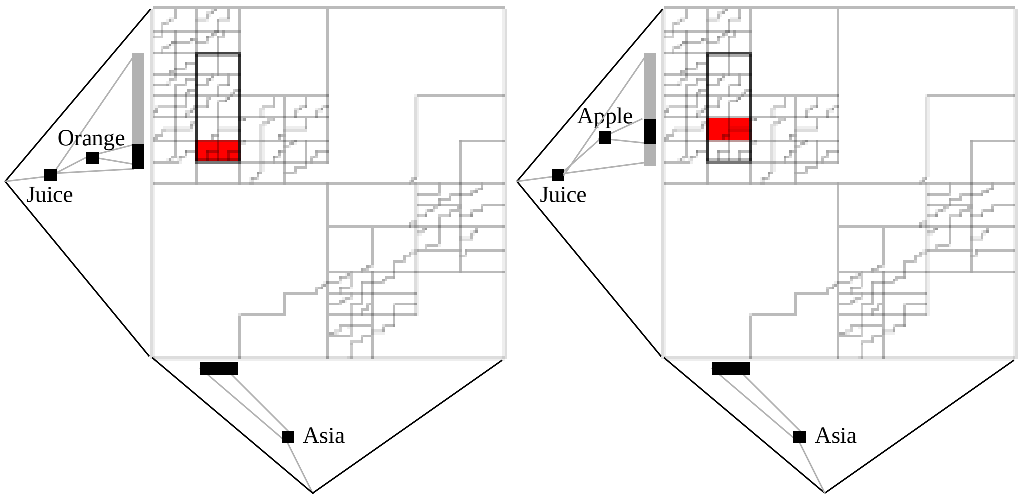


Figure 5-9: Processing a query box in sort order with the Tetris algorithm

5.3.4.7 Materializing Aggregates

Multidimensional hierarchical clustering is not only applicable to the raw data itself, but can also be used to organize views with materialized aggregates. Higher aggregation levels result in a UB-Tree with shorter compound surrogates or reduced dimensionality. It makes sense to store pre-computed aggregates for the highest aggregation levels with restrictions in only one dimension, e.g., the total sales on a yearly basis. However, multidimensional hierarchical clustering allows one to derive many aggregates efficiently from the raw data. This avoids materialization of many aggregation levels and thereby reduces the view maintenance problem for summary tables to a large extent.

5.3.5 Dealing with non-uniformly distributed Data with Quantiles

In Section 3.10 we identified the puff pastry effect as a severe problem of UB-Trees for certain data distributions. The problem is due to the fact that UB-addresses are always calculated by splitting an attribute in the center of a subcube. The space partitioning can be improved considerably by taking the actual domain of each attribute into account. More precisely, if the attribute is not split in the middle of the domain, but in the middle of the data distribution, the space partitioning is greatly improved with respect to spatial proximity. This basically means that α-quantiles (i.e., the value of a domain where the cumulated data distribution exceeds α, see e.g., [Zwi96]) are used to identify the Cartesian coordinate of each subdivision point α. Quantiles were previously used in combination with hashing techniques [KS87]. The *variable UB-Tree* is an approach which combines quantiles with UB-Trees.

5.3.5.1 The Variable UB-Tree

Definition 5-8 (variable UB-Tree, VUB-Tree): A *variable UB-Tree* (VUB-Tree) is a UB-Tree, whose addresses are calculated by subdividing each dimension with respect to the quantiles according to the data distribution of this dimension.

VUB-Trees recursively subdivide the embedding space with respect to the data distribution. Thus for each dimension the first subdivision step compares each attribute value to the value for which the cumulated data distribution of this dimension exceeds 50% (50%-quantile). Accordingly, the two values for the second subdivision step are the values, where the cumulated distribution exceeds 25% and 75% respectively (25%-quantile, 75%-quantile). Consequently, we will call these subdivision points split points. Note that for uniformly distributed data the split points of a VUB-Tree are identical to those of a UB-Tree.

VUB-Trees therefore use a transformation function to re-distribute the values of a domain. The resulting bit strings are equally distributed. This means, that with variable UB-Trees the behavior of UB-Trees for uniformly distributed data is achieved for any data set which is distributed independently in the dimensions.

Figure 5-10 illustrates how the split points for various data distributions are derived. The thick line at 50% of the cumulated distribution calculates the split point for step 1, the two thinner lines at 25% and 75% calculate the split points for step 2, and the even thinner lines at 12.5%, 37.5%, 62.5% and 87.5% calculate the split points for step 3. Figure 5-10a displays polynomially distributed data. Some skewed data distribution is shown in Figure 5-10b. Figure 5-10c shows uniformly distributed data, which results in recursively halving the domain and thus yields the split points of non-variable UB-Trees.

The address calculation algorithm of VUB-Trees is just a slight modification of the UB-Tree address calculation. Values are now compared to split points instead of fixed subdivision points. Thus VUB-Trees use a special transformation function that calculates the bit string of an attribute with respect to the data distribution. The conventional bit interleaving algorithm is then used to calculate the addresses of the VUB-Tree.

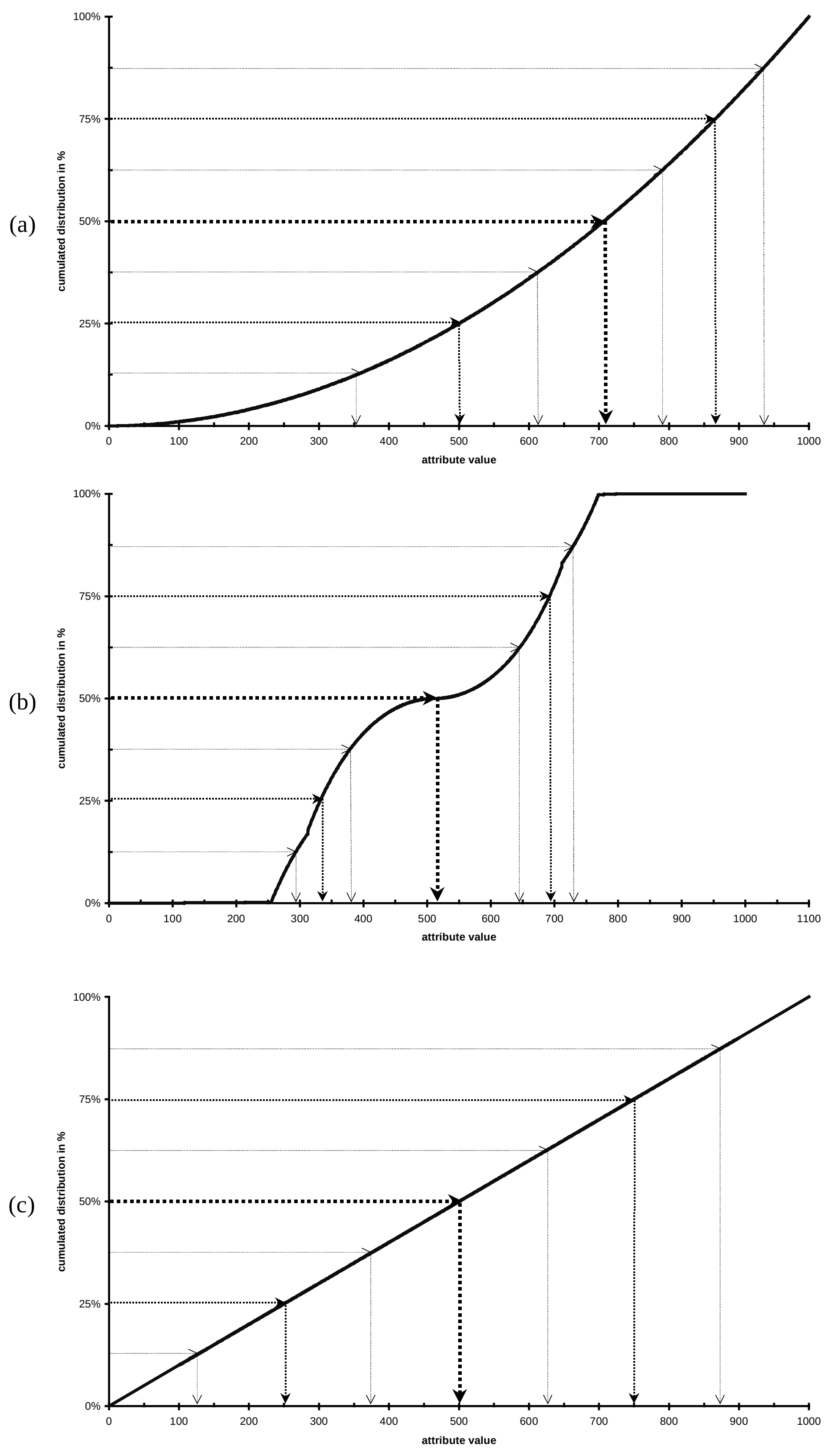


Figure 5-10: Calculation of split points for various data distributions

VUB-Trees yield a better space partitioning for non-uniformly distributed data. The transformation function performs a re-distribution of data. Uniformly distributed attribute bit strings are created from the data of any data distribution. Thus common prefixes do not exist for any attribute bit string. This avoids the puff pastry effect described in Section 3.10.

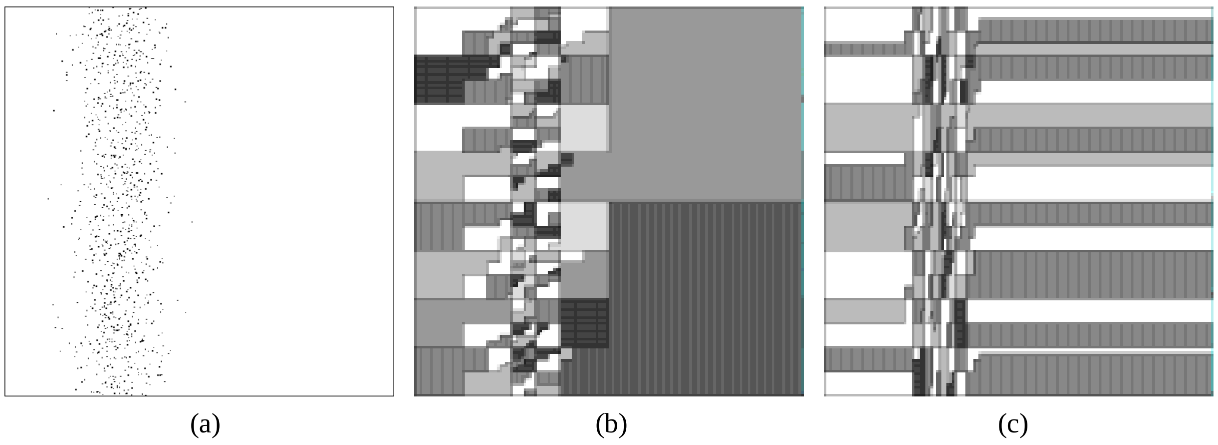


Figure 5-11: UB-Tree and VUB-Tree for Gaussian/uniform data distribution

Figure 5-11a shows data which is distributed uniformly in the vertical dimension and Gaussian in the horizontal dimension. The corresponding UB-Tree in Figure 5-11b shows a strong puff pastry, since the split points in the center of the space are not useful for the horizontal dimension. In contrast to that the VUB-Tree in Figure 5-11c shows an equal number of splits in both dimensions.

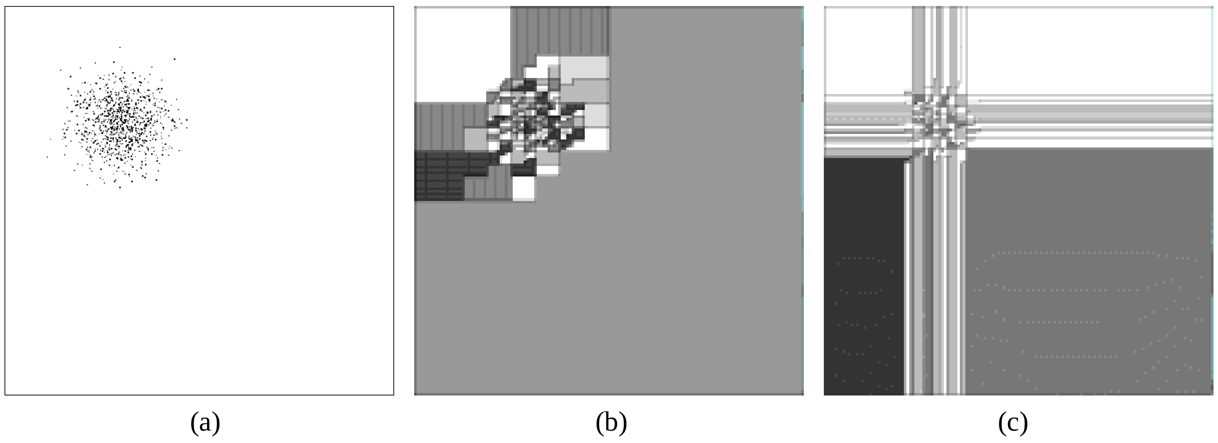


Figure 5-12: UB-Tree and VUB-Tree for Gaussian data distribution

Figure 5-12a shows data which is Gaussian distributed in both dimensions. Again the corresponding UB-Tree (Figure 5-12b) yields a puff pastry, which in this case is not so strong as in Figure 5-11b, since both dimensions have common prefixes. Since these prefixes do not have identical length, the VUB-Tree (Figure 5-12c) again yields a better partitioning than the UB-Tree.

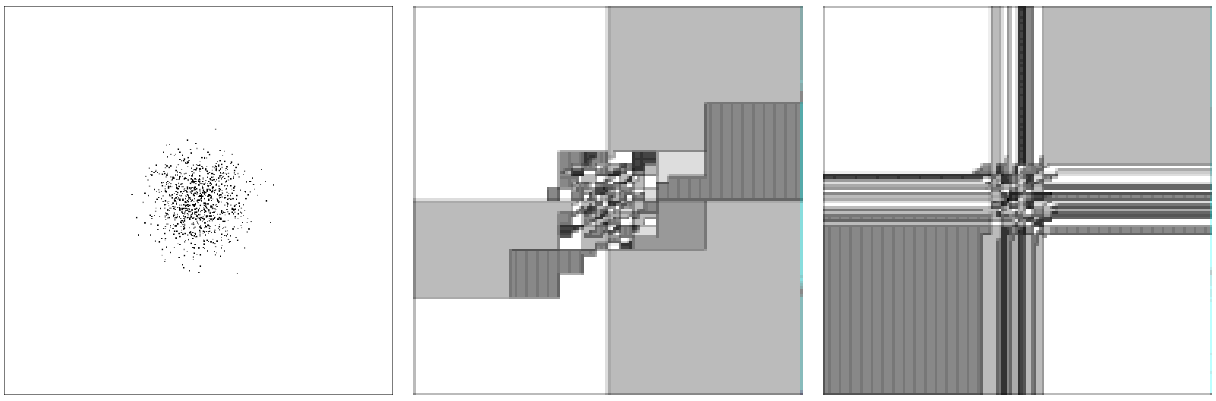


Figure 5-13: UB-Tree and VUB-Tree for centered Gaussian data distribution

Figure 5-13a shows data with Gaussian distribution in both dimensions. Here the average and the standard deviation are identical for both dimensions. Therefore the partitioning does not yield a puff pastry (Figure 5-13b). However, variable UB-Trees produce a different space partitioning (Figure 5-13c). Figure 5-12 and Figure 5-13 show that for both UB-Trees and VUB-Trees the location of the data distribution does not matter, if the prefixes in all dimensions are identical (see also Section 3.10).

5.3.5.2 Split Point Trees

Definition 5-9 (split point tree): A split point tree is a binary tree that stores the split point hierarchy. The root of the tree consists of the values where the cumulated distribution of each dimension exceeds ½. The left and right son store the values where the cumulated distributions exceed ¼ respectively ¾ (25%-quantile, 75%-quantile). In general a split point tree of height h stores 2^h split points corresponding to the cumulated distribution in discrete steps of $k \cdot 2^{-h}$ for $k \in \{1, ..., 2^h\text{-}1\}$.

A split point tree is an efficient way of storing the split points for each independent dimension of a variable UB-Tree. Assuming a split point tree of height h, a dimension then can contribute h bits to the VUB-address. The total length of each address then is $d{*}h$. Databases consisting of $p = 2^{d*h}$ pages can be managed with a split point tree of height h. Thus, the height of the split point tree for each dimension is $h = \left\lceil \frac{\log_2 P}{d} \right\rceil$, which is quite small compared to the size of the database. For instance, a split point tree of height 4 for a 6-dimensional database is sufficient for a database size of up to 2^{24} pages. Table 5-4 lists the height of a split point tree for databases with 1 million, 10 million and 100 million data pages for 2 to 10 dimensions.

	2d	3d	4d	5d	6d	7d	8d	9d	10d
1 mio. = 200 MB	10	7	5	4	3	3	2	2	2
10 mio. = 2 GB	12	8	6	5	4	3	3	3	2
100 mio. = 20 GB	13	9	7	5	4	4	3	3	3

Table 5-4: Heights of a Split Point Tree for various Database Sizes and Dimensions

5.3.5.3 Implementation of Variable UB-Trees

To implement variable UB-Trees, only the transformation function of the address calculation algorithm needs to be modified. A split point tree is used to determine the bit string of an attribute. Each decision in the split point tree decides to which part of the data distribution a point belongs.

VUB-Trees were implemented by Michael Bauer in a master thesis [Bau98] supervised by the author. A more detailed description of split point trees and the algorithms involved can be found there.

Since VUB-Trees just change the attribute transformation, no modifications of other algorithms like range query, point query, or Tetris, are necessary.

However, VUB-Trees have two major drawbacks:

- The data distribution for each attribute must be known in advance.
- Dimensions must be independent.

Split point trees require the data distribution to be known in advance. This is possible, if statistics on the data are available or expectations on the data exist. For OLAP data another efficient method is to use a bulk loading tool. A VUB-Tree can be built by two passes over the data. The first pass gathers the statistics and the second pass calculates the addresses for each tuple. If the data distribution changes, the entire VUB-Tree needs to be re-organized. Otherwise the VUB-address space might be exhausted, which causes the multidimensional clustering to be no longer useful, since many tuples will be addressed by the same address.

The dimensions of a VUB-Tree must be independent. For certain dependencies like the sine dependency the VUB-Tree cannot avoid the puff-pastry effect or run into problems with the height of the split point tree (see [Bau98]).

5.4 Implementing the Insertion Algorithm

The complexity of the insertion algorithm is logarithmic in the table size and relies essentially on the implementation of the point query algorithm. Since the implementation of the point query algorithm is discussed in Section 4.3, we focus on the Z-region split algorithm here.

Figure 5-14a shows that a Z-region partitioning usually does not subdivide a multidimensional space into rectangular Z-regions, but usually consists of Z-regions which are a union of several rectangular subcubes. We say that such a region has *fringes*. Thus fringes are a union of small subcubes which in addition to the largest subcube belong to one region.

Fringes cannot be avoided in general, since the Z-region partitioning adapts to the data distribution as shown in sections 3.7 and 3.8. However, it is desirable to minimize fringes, since a small fringe might cause a Z-region to overlap a query box. This Z-region then must be retrieved by the range query algorithm, although probably no points or only very few points on the Z-region are located in the query box.

A Z-region is split due to an overflow of the corresponding page. The Z-region split then creates two Z-regions from the Z-region being split. Minimizing fringes thus can be achieved by trying to create rectangular regions during the split. This may be achieved by giving this algorithm some more flexibility. The split algorithm consists of two parts:

- choose two points on the page not exactly in the middle, but close around the middle (ε-split)

- find the split address between two points that yields the “best” rectangular partitioning

The “best” rectangular partitioning of a Z-region $[\alpha : \beta]$ is calculated by choosing a split address ξ where as many trailing bits as possible are set to 1. In the following we assume that a page is an ordered set of tuples $\{x_1,...,x_C\}$. The split algorithm determines the Z-address ξ with the most trailing bits between the Z-addresses of the tuples $x_{½\cdot C-\varepsilon\cdot C}$ and $x_{½\cdot C+\varepsilon\cdot C}$. Depending on ε, fringes are avoided to a large extent. A worst case storage utilization of 50%- ε is guaranteed for each page. For our tests with uniformly distributed data fringes get reduced to a large extent. With an ε of 5% the partitioning of uniformly distributed data gets very close to a uniformly idealized partitioning. Identical uniformly distributed data was spooled into the UB-Trees of Figure 5-14a, b and c. While the UB-Tree of Figure 5-14a only takes the tuple in the middle of the page into account for a region split, Figure 5-14(b and c) use an ε of 1% respectively 40%. The picture clearly shows that fringes are minimized with growing ε at the expense of a reduced storage utilization. As we will see in Section 6.1.5, the range query performance is also improved by using an $\varepsilon > 0$. Our measurements showed that an ε of a few percent already has a substantial effect on the range query performance. For a detailed investigation of the effects of the ε–split please refer to [Sch98].

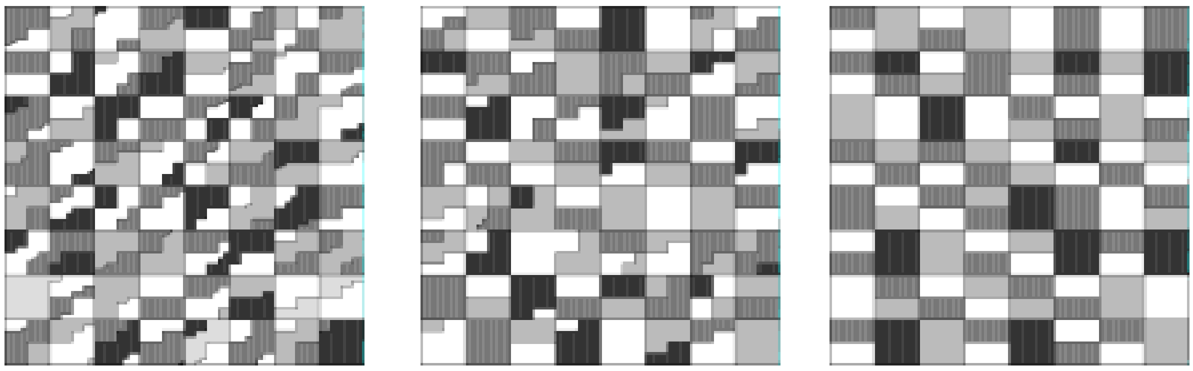


Figure 5-14: Splitting Z-regions with ε = 0%, ε = 1% and ε = 40%

5.5 Implementing the Deletion Algorithm

The deletion algorithm works analogously to the deletion algorithm for B-Trees [BM72]. However, when a deletion causes a page underflow and the corresponding page merge causes an overflow, the subsequent page split utilizes the splitting algorithm of Section 5.4 to create a "good" space partitioning. See [Sch98] for details.

5.6 Implementing the Point-Query Algorithm

The point query algorithm for a tuple *x* reads a single Z-region $[\alpha : \beta]$ and the corresponding page from a UB-Tree. The region address β is defined as the nearest upper neighbor in the UB-Tree index table to the Z-address ξ of tuple *x*. As already mentioned in Section 2.2.4, this nearest neighbor is found by a single ESQL query. This region address then is used by a second ESQL query to retrieve the entire data page.

```
ESQL SELECT MIN(z-address) INTO region_address
    FROM UB-Tree_index_table
    WHERE z-address ≥ ξ
ESQL SELECT number_of_tuples, data_page_content INTO tuples, page
    FROM UB-Tree_index_table
    WHERE z-address = region_address
```

The performance of the first ESQL query heavily relies on the DBMS optimizer. In general queries with an aggregation function and a restriction are evaluated by first retrieving all tuples satisfying the restriction and then applying the aggregation function on the retrieved tuple set. However, if a B^*-Tree index on *z-address* exists, the optimal execution plan for this statement is different: The combination of the aggregation function MIN() with a ≥-restriction in the WHERE-clause allows one to determine the result by traversing just one path in the B^*-Tree: A point query with ξ yields the page that stores the region address as the first value exceeding ξ. However, only the TransBase optimizer created this optimal execution plan.

Figure 5-15 shows the access plans of Oracle (a) and TransBase (b) for the SELECT-MIN-query. Figure 5-15c shows the actual operation that TransBase performs on the UB-Tree secondary index table in order to find the region address.

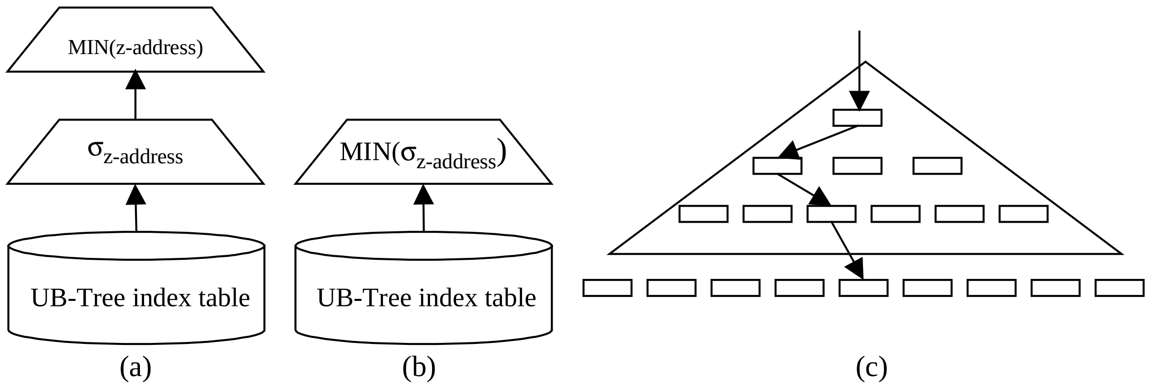


Figure 5-15: Implementation of the point query algorithm

For DB2 and Oracle it was necessary to rewrite the query in the following way:

```
ESQL DECLARE CURSOR minquery FOR
    SELECT z-address from UB-Tree_index_table
        WHERE z-address ≥ ξ
        ORDER BY z-address
ESQL OPEN CURSOR minquery
ESQL FETCH minquery INTO region_address
ESQL CLOSE CURSOR minquery
ESQL SELECT number_of_tuples, data_page_content INTO tuples, page
    FROM UB-Tree_index_table
    WHERE z-address = region_address
```

The ORDER BY clause of the rewritten query forces the optimizer to favor a secondary index on *Z-address* over a full table scan. Since Z-addresses are delivered in ascending order by the query, the first FETCH returns the minimum *Z-address*. Therefore the rewrite is equivalent to the original query.

5.7 Implementation of the Range Query Algorithm

The range query algorithm as described in Section 4.5 is exponential in the number of dimensions. A linear version of this algorithm was developed by the author. In addition the algorithm was optimized to operate solely on Z-addresses, which saves CPU time for transformations between Z-addresses and Cartesian tuples. The main function of the range query algorithm is the function which, for a given Z-region, determines the next Z-region intersecting the query box. Next to the Z-addresses of start point y and end point z of a query box $Q = [[y, z]]$ the region address β of a Z-region $[\alpha : \beta]$ suffices to perform the calculation.

5.7.1 Determining the next Intersection

The crucial task of the range query algorithm is to calculate the next region in Z-order which intersects the query box Q after a Z-region $[\alpha : \beta]$ (which intersects the query box) has been retrieved. This algorithm has been steadily refined in the MISTRAL project during the evolution of this thesis. While the first version of the algorithm was exponential in the number of dimensions and required a transformation of Z-addresses into Cartesian co-ordinates for each iteration, the final version merely requires $O(a)$ bit operations (set bit and clear bit) for standard Z-addresses consisting of a bits.

The *exponential version* of the algorithm has been described in Section 4.4. In principle, this algorithm is divided into two steps:

- determination of the Z-address λ of the last point of the Z-region $[\alpha : \beta]$ intersecting the query box
- determination of the Z-address ϕ of the next Z-region intersecting the query box

Since each subdivision step divides the multidimensional space into 2^d subcubes, this algorithm is exponential in the number of dimensions. In addition, it requires a transformation of Z-addresses into Cartesian co-ordinates in order to test the intersection of the query box with the subcube. The overall CPU-complexity (in address transformations, integer comparisons and bit-operations) of this algorithm is $O(a^2/d \cdot 2^d)$.

A *linear version* of this algorithm can be developed, observing that dimensions are independent and query boxes are iso-oriented. Thus the intersection can be tested for each dimension independently. Then, to construct the Z-address ϕ it is not necessary to traverse 2^d subcubes for each step of ϕ and to check if that subcube intersects the query box. Instead, it suffices to test if dimension j intersects the interval $[y_j, z_j]$. Thus only d tests are necessary. However, still necessary is a transformation of the address to Cartesian coordinates for each test, thus the overall complexity is $O(a/d \cdot d \cdot a/d) = O(a^2/d)$.

The linear algorithm can be further improved into a *bit-oriented algorithm* by transforming the co-ordinates y and z into addresses ψ and ζ. Then a transformation of the current Z-addresses to Cartesian coordinates is not necessary to perform the intersection calculation; the intersection calculation is solely done on the Z-address representation. (It is possible to implement an intersection test on Z-addresses, which determines by simple bit operations if a Z-address is located in a query box defined by two Z-addresses [MOD99]). Each bit of a Z-address is inspected at most twice by the bit-oriented algorithm in order to determine the first Z-address of the next Z-region intersecting the query box. Thus the overall complexity of the algorithm is $O(a)$.

Additional details about the implementation of the range query algorithm can be found in the MISTRAL Source Code Documentation [MOD99]. Several experiments were performed to evaluate the practicability of the algorithm [Fri97]. The major results of these experiments were:

- The exponential algorithm is easy to implement; however, it is useful only for up to three dimensions.
- The linear algorithm is useful for up to 6-dimensional Z-addresses, if the result set size is sufficiently small. Otherwise the overhead for the Z-address/Cartesian transformation plays a significant role.
- The bit oriented algorithm is insensitive to the complexity of the tuple transformation. Therefore UB-Trees using complex tuple transformations in order to create a good space partitioning (like the surrogate calculation (Section 5.3.3 and 5.3.4) or variable UB-Trees (Section 5.3.5)) can be built without additional cost on the range query algorithm.

5.7.2 Dealing with Sets of Query Boxes

Retrieving a result set defined by a set of query boxes $\mathbb{Q} = \{Q_1, ..., Q_n\}$ is possible by only accessing each page of the result set once. This is achieved by calculating the first intersection ϕ_i for each Q_i as mentioned in Section 5.7.1. These intersections $\{\phi_1, ..., \phi_n\}$ are stored in a heap $\mathbb{H}$ sorted in Z-order. MIN($\mathbb{H}$) then determines the next Z-region $[\alpha : \beta]$ to be retrieved by the range-query-set algorithm. Then all ϕ_i with $\phi_i \leq \beta$ can be updated in $\mathbb{H}$ by removing these ϕ_i and storing a new ϕ_i' calculated by the algorithm of Section 5.7.1.

This algorithm requires space O(n) for n query boxes and has a CPU-time complexity of O($a \cdot n$) for Z-addresses consisting of a bits. The I/O-complexity of the algorithm is O(m) (plus the overhead for the B-Tree access to each page) if m regions overlap $\mathbb{Q}$. Further details on the implementation of the algorithm as well as performance measurements are described in [Fen98]. In addition, investigations about the approximation of arbitrary query volumes can be found there. Here we just sketch the main results:

- Dealing with sets of query boxes induces no additional CPU- and I/O-overhead to the range query algorithm. Additional main memory in O(n) is required to process a set of n query boxes.
- In the case of one query box the algorithm degenerates to the standard range query algorithm.
- Disk accesses are performed in Z-order, so if the relation is page clustered, this page clustering will be exploited by the algorithm.
- When sets of query boxes are used to approximate arbitrary query volumes, for uniformly distributed data the best performance is achieved when the volume of each query box gets close to the average volume of each Z-region.

5.8 The UB-Tree Library

The UB-Tree library consists of an API and an operator interface. The UB-API provides a C-interface for data manipulation and data definition functions. The functionality of the UB-API includes C-functions for

- UB_create, UB_drop, UB_rename
- UB_open, UB_close
- UB_insert, UB_delete
- UB_pointquery, UB_rangequery

While UB-API functions deal with result sets of tuples which are materialized in one processing step, the operator interface deals with streams of tuples. The operator interface essentially provides the same functionality as the UB-API. However, since the operator interface works with tuple streams and allows pipelined processing, it is better suited for using the UB-Tree functionality in an operator tree of an RDBMS. Next to the functions listed above a preliminary version of the Tetris algorithm for uniformly distributed data is implemented with the operator interface.

The UB-API has a tuple handling which allows one to use signed integer, unsigned integer and character strings for the calculation of Z-addresses. The concept of transformation functions has been implemented to easily plug in support for further data types. The current implementation allows one to calculate Z-addresses for UB-Trees with up to 31 index dimensions.

A detailed description of the UB-Tree library is found on the MISTRAL web server [MOD99].

Part III

Analysis of Query Processing with Multidimensional Indexes

Ideals are like the stars, we never reach them, but like the mariners of the sea, we chart our course by them.

(Carl Schurz)

Chapter 6

Performance Analysis

Analyzing the cost of query processing with certain access methods is crucial to cost-based query optimization. Next to that it allows one to simulate query processing without actually creating the database and thus saves time and resources while gaining a better understanding of access methods. In addition a cost function is a benchmark for the query performance since it defines the expected response time and thus allows one to judge the quality of an access method implementation. Cost functions can further be used to predict the result sizes of a query or tell a user the expected processing time of a query before query execution has started. Since multi-attribute restrictions and sort operations are very common in many applications (cf. Section 1.1), we in this chapter investigate the cost of range queries and the Tetris algorithm for UB-Trees. We define cost functions for page accesses for idealized uniformly partitioned universes. The quality of our cost function is proven by comparing the predicted number of page accesses with the actual number of page accesses measured with our prototype implementation. We also explain how our cost function is linked to the selectivity of a multidimensional query box. Then we define a cost model for range queries with sort operations for various further access methods and do an analytical performance comparison of UB-Trees with bitmap indexes, clustering B-Trees, non-clustering B-Trees and a full table scan. Chapters 7 and 8 will show that the results predicted by our cost functions do also hold in practice.

6.1 The Cost of UB-Tree Range Queries

For an analysis of the UB-Tree range query performance it is desirable to have a cost function for the retrieved pages, i.e., the regions overlapped by a query box. This enables the prediction of the run time of a range query and yields a base for query optimization. In addition, a cost function permits to produce a statistical relevant number of measures by simulating range queries with varying table sizes and dimensionalities. This is especially useful, when practical measurements with our pilot implementation can not be performed due to their memory requirements or their long run time. The cost function also allows a theoretical analysis of the range query performance and provides an excellent insight in the importance of the attribute order. It also explains the page jump phenomenon of UB-Tree range queries. Several master students supervised by the author used this cost function to simulate the behavior of the UB-Tree for several database sizes and dimensionalities [Fri97], to analyze the effect of region fringes and to judge the quality of the region splitting strategy [Sch98], to analyze the space partitioning of VUB-Trees [Bau98] and to analyze the behavior of UB-Trees for sets of query boxes [Fen98].

To be useful for a broad range of applications, a cost function should not require too much knowledge about the queried database. The minimum requirements for input parameters of a cost function are the database size and the query restriction. For multi-dimensional index structures the dimensionality of the index is also necessary. These parameters are in general easy to obtain and maintain.

To derive a cost function using only these input parameters is not achievable in general, since the data distribution is another decisive factor for the query performance. Yet it is possible to develop such a cost function for a certain case of space partitioning, the so-called idealized uniform partitioning, consisting of uniformly distributed and independent attributes.

6.1.1 A Cost-Function for Perfect Idealized Uniform Partitioning

In the following we use P for the number of data pages of a UB-Tree table. The query box is specified by the lower bounds vector y and the upper bounds vector z. The lower bound and upper bound of the restriction in dimension i are denoted by y_i and z_i. The UB-Tree has been built over a total of d dimensions.

Definition 6-1 (perfect idealized uniform partitioning): A *perfect idealized uniform partitioning* (i.e., $P = 2^{d \cdot k}$ for some $k > 0$, cf. Figure 6-1a for a two-dimensional example) subdivides the multidimensional space in P hypercubes with volume $2^{-d \cdot k}$. Otherwise we call an idealized uniform partitioning *imperfect*.

Thus for perfect idealized uniform partitioning each dimension j of the universe has been partitioned recursively $l_j(d, P) = \log_2 P \,/\, d = k$ times. In this case the number of Z-regions intersected by a query box [[y, z]] is identical to the number of subcubes overlapped by [[y, z]].

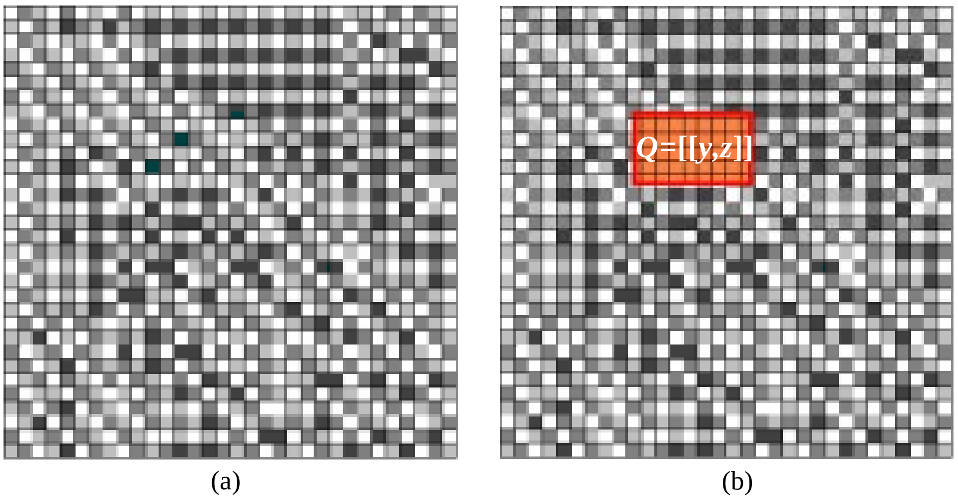


Figure 6-1: The cost function for perfect idealized uniform partitioning

In the following we assume the boundaries of the query interval in each dimension to be normalized to the interval [0,1].

If we have *s* completed split levels in dimension *j*, the number of slices of the multi-dimensional space that are overlapped by the query box [[*y*, *z*]] in dimension *j* can be determined by calculating the number of the slices, that are contained in interval [0, $\hat{z}_j$], but not in interval [0, $\hat{y}_j$], i e., the number of slices in [0, $\hat{z}_j$] minus the number of slices in [0, $\hat{y}_j$]. Since the slice containing $\hat{y}_j$ is also a slice overlapped by the query box, we must increment the above number by one to get the correct number of slices. If the number of slices for the interval [0, $\hat{c}$] is calculated as $\lfloor \hat{c} \cdot 2^s \rfloor$, a non-existing slice 2^s+1 is added for $\hat{z}_j=1$ by the formula derived above. We must correct this error for the case $\hat{y}_j < 1$ and $\hat{z}_j = 1$. This is achieved by decrementing the number of slices by one. For $\hat{y}_j = \hat{z}_j = 1$ the subtraction removes the error, therefore no correction is necessary here.

Thus the number of slices $n(y_j, z_j, l_j)$ in dimension *j* overlapped by the query interval $[y_j, z_j]$ for l_j completed splits in dimension *j* can be calculated by the following formula:

$$n(y_j, z_j, l_j) = \begin{cases} 2^{l_j} - \lfloor \hat{y}_j 2^{l_j} \rfloor & \text{, if } \hat{z}_j = 1 \wedge \hat{y}_j \neq 1 \\ \lfloor \hat{z}_j 2^{l_j} \rfloor - \lfloor \hat{y}_j 2^{l_j} \rfloor + 1 & \text{, otherwise} \end{cases}$$

The number of subcubes intersected by the query box $c(y, z, P, d)$ then is the product of $n(y_j, z_j, l_j(d, P))$ over all dimensions:

$$c(d, P, y, z) = \prod_{j=1}^{d} n\left(y_j, z_j, l_j(d, P)\right)$$

6.1.2 A Cost Function for Semi-perfect Idealized Uniform Partitioning

An imperfect idealized uniform partitioning produces rectangular regions, where each region has either the shape of a subspace with volume $2^{-\lceil \log_2 P \rceil}$ or consists of two of these subspaces. In this case the multidimensional space has been partitioned recursively $\lfloor \log_2 P \rfloor \bmod d$ times for some dimensions and $\lfloor \log_2 P \rfloor \bmod d + 1$ times for some other dimensions. One dimension may exist, where some parts of the space already have been partitioned $\lfloor \log_2 P \rfloor \bmod d + 1$ times, while other parts of the space only have been partitioned $\lfloor \log_2 P \rfloor \bmod d$ times.

Because of the above considerations we distinguish two cases of imperfect partitioning:

Definition 6-2 (semi-perfect and probabilistic idealized uniform partitioning): If $P = 2^k$ for some $k > 0$, we call an imperfect idealized uniform partitioning *semi-perfect*. Otherwise we call it *probabilistic*.

Definition 6-3 (probabilistic dimension): For a probabilistic idealized uniform partitioning we call a dimension *probabilistic*, if with respect to this dimension some parts of the space have been partitioned $\lfloor \log_2 P \rfloor \bmod d + 1$ times, while other parts of the space only have been partitioned $\lfloor \log_2 P \rfloor \bmod d$ times.

Lemma 6-1: Each probabilistic idealized uniform partitioning has exactly one probabilistic dimension.

Proof:

Bit interleaving takes place in a fixed order of dimensions. Our implementation of bit interleaving starts with the rightmost dimension. 2^l splits need to take place to completely split the space with respect to split level l (i.e., bit l of the binary representation of the Z-address). After these 2^l splits have taken place (i.e., enough data has been inserted into the UB-Tree), the next split takes place at split level $l + 1$ (i.e., bit $l + 1$ of the binary representation of the standard address). This split level corresponds to the next bit in the binary representation of standard addresses as obtained by bit interleaving. Therefore it splits the next dimension in the order of dimensions as used by bit interleaving.

Since splits complete one split level before moving to the next split level, only one dimension may have both subspaces with split level l and subspaces with split level $l + 1$. □

As a consequence of the proof of Lemma 6-1 the index of the probabilistic dimension in the order of dimension as used by bit interleaving (Algorithm 5-1) is calculated as:

$$\text{probabilistic}(d, P) = d - \left(\lfloor \log_2 P \rfloor \bmod d \right)$$

Example 6-1:

Split levels for perfect and imperfect uniform partitioning are illustrated in Table 6-1 for a 6-dimensional space: For a table size of 64 pages the space is perfectly partitioned (k = 1, d = 6) with one split level for each dimension. With 512 (k = 9) pages this space is partitioned semi-perfectly with one split level in the first three dimensions and two split levels for the last three dimensions. With a page number of 700, the partitioning is probabilistic with dimension 3 as probabilistic dimension.

Pages	Split Levels per Dimension					
	Dim 1	Dim 2	Dim 3	Dim 4	Dim 5	Dim 6
64 (perfect)	1	1	1	1	1	1
512 (semi-perfect)	1	1	1	2	2	2
700 (probabilistic)	1	1	>1	2	2	2

Table 6-1: Split levels

For a semi-perfect idealized uniform partitioning the number of completed splits $l_j(d,P)$ with respect to dimension j is calculated as

$$l_j(d,P) = \begin{cases} l_{j\downarrow}(d,P)+1 & \text{,if } \lfloor \log_2 P \rfloor \bmod d \le j \\ l_{j\downarrow}(d,P) & \text{,otherwise} \end{cases} \quad \text{where } l_{j\downarrow}(d,P) = \left\lfloor \frac{\log_2 P}{d} \right\rfloor$$

With $l_j(d,P)$ as defined above the $c(y, z, P, d)$ is calculated in the same way as for perfect idealized uniform partitioning:

$$n(y_j, z_j, l_j) = \begin{cases} 2^{l_j} - \lfloor \hat{y}_j 2^{l_j} \rfloor & \text{,if } \hat{z}_j = 1 \wedge \hat{y}_j \neq 1 \\ \lfloor \hat{z}_j 2^{l_j} \rfloor - \lfloor \hat{y}_j 2^{l_j} \rfloor + 1 & \text{, otherwise} \end{cases}$$

The key attributes are independent and the query box $[[x, y]]$ is iso-oriented with respect to each dimension. Therefore the total number of pages is obtained by multiplication of the slices in each dimension:

$$c(d,P,y,z) = \prod_{j=1}^{d} n\left(y_j, z_j, l_j(d,P)\right)$$

6.1.3 A Cost Function for Probabilistic Idealized Uniform Partioning

If an idealized uniform partitioning is probabilistic, the formula $n(y_j, z_j, l_j(d, P))$ needs to be modified for the probabilistic dimension to take the probability of an incomplete split into account.

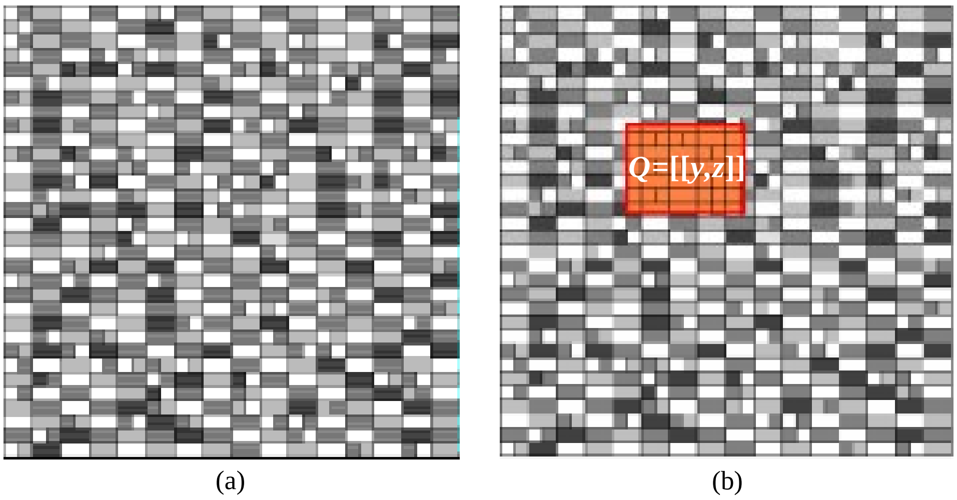


Figure 6-2: The cost function for probabilistic uniform partitioning

The complete split levels produce only $2^{\lfloor \log_2 P \rfloor}$ pages, thus P - $2^{\lfloor \log_2 P \rfloor}$ additional regions are needed to obtain the given number of pages, i.e., the table size. These regions are created from the $2^{\lfloor \log_2 P \rfloor}$ pages by splitting these pages with respect to attribute j. Therefore the probability of an additional split in an attribute j is:

$$\text{probability}_j(d,P) = \begin{cases} \dfrac{P}{2^{\lfloor \log_2 P \rfloor}} - 1 & \text{, if } j = \text{probabilistic}(d,P) \\ 0 & \text{, otherwise} \end{cases}$$

If the probability of an incomplete split is taken into account, the number of slices overlapped by a query range in a certain dimension can be derived from the value for the completed splits. By subtraction we calculate, how many slices would be overlapped additionally, if another completed split were existing. For each of these splits the probability of its existence is $\text{probability}_j(d, P)$. The average number of additional splits may then be calculated by multiplication.

Thus, the average number of slices in dimension j overlapped by the range $[y_j, z_j]$ is:

$$n_j(d, P, y_j, z_j) = n(y_j, z_j, l_j(d, P)) + (n(y_j, z_j, l_j(d, P) + 1) - n(y_j, z_j, l_j(d, P))) \cdot \text{probability}_j(d, P)$$

The key attributes are independent and the query box $[[x, y]]$ is iso-oriented with respect to each dimension. Thus the total number of pages is obtained by multiplication of the slices in each dimension as in the previous sections.

6.1.4 Cost Function and Selectivity

For independently uniformly distributed data the restriction of each attribute in percent of space also defines the selectivity of that attribute. Thus the restriction $[y_j, z_j]$ in attribute A_j has a selectivity of $s_j = \hat{z}_j - \hat{y}_j$.

$\prod n_j(d, P, y_j, z_j)$ can be considered to be $\prod \hat{s}_j \cdot P$ with $\hat{s}_j$ as a special ceiling function rounding the selectivity s_j of dimension j to the next partitioning grid point. Then the cost function can also be represented as:

$$c(d,P,s) = \prod_{j=1}^{d} n_j(d,P,0,s_j) = P \cdot \prod_{j=1}^{d} \hat{s}_j$$

Note that by using 0 and s_j instead of z_j and y_j some information for the accuracy of the cost function is lost, since the position of the query box influences the number of Z-regions overlapped by a query box. Thus $c(d, P, y, z)$ should be used instead of $c(d, P, s)$, if not only the selectivity, but also start and endpoint of the query in space are known.

6.1.5 The Cost Function: Theory and Practice

Figure 6-3 shows the number of pages predicted by the cost function as well as the actually retrieved number of pages for two range query series. Both measurement series where conducted on a six dimensional UB-Tree of 211218 pages storing 10 million uniformly distributed tuples. The left series shows a query that constantly restricts five dimensions to 40% of their domain and varies the sixth dimension from 1% to 100% of its domain. The right series varies the restriction in each dimension from 1% of its domain to 100% of its domain at the same time. Thus the left series creates a linearly growing result set, whereas the result set of the right series grows with the 6th power.

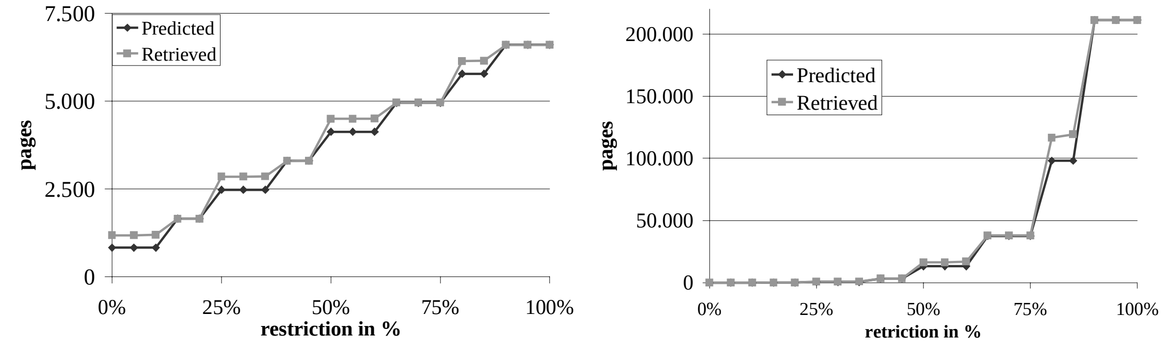


Figure 6-3: Cost function and reality

Both series show the accuracy of the cost function with an average prediction error of 8%. The maximum deviation is 22% in the left series and 30% in the right series.

The deviation is due to two facts:

- The data is just distributed uniformly, but there is actually no idealized uniform partitioning.
- The model is probabilistic, thus a certain error is inherent to the cost model.

The effect of the ε-splitting algorithm (cf. Section 5.4) on the exactness of the cost function is shown in Figure 6-4. The six dimensional UB-Tree of this figure consists of 1 million tuples of uniformly distributed data stored on about 22000 pages (the actual number of pages differs for each ε, since the degree of filling for each page depends on ε). The figure shows the actually retrieved pages for various values of ε as well as the theoretically expected value from the cost function for a query which restricts five dimensions to 15% of their domain and varies the sixth dimension from 1% to 100%. The larger ε gets, the better the region partitioning approximates an idealized uniform partitioning. The figure therefore shows the accuracy of the cost function and the potential of the region splitting algorithm to reduce the number of pages overlapped by a range query. In general a trade-off between space partitioning (i.e., range query performance) and page utilization exists. However, as the figure shows, an ε of around 3% is already quite effective. Due to the probabilistic nature of the cost function an ε of 100 % sometimes even retrieves less pages then predicted. For more details to the ε-splitting algorithm please refer to sections 4.2 and 5.4.

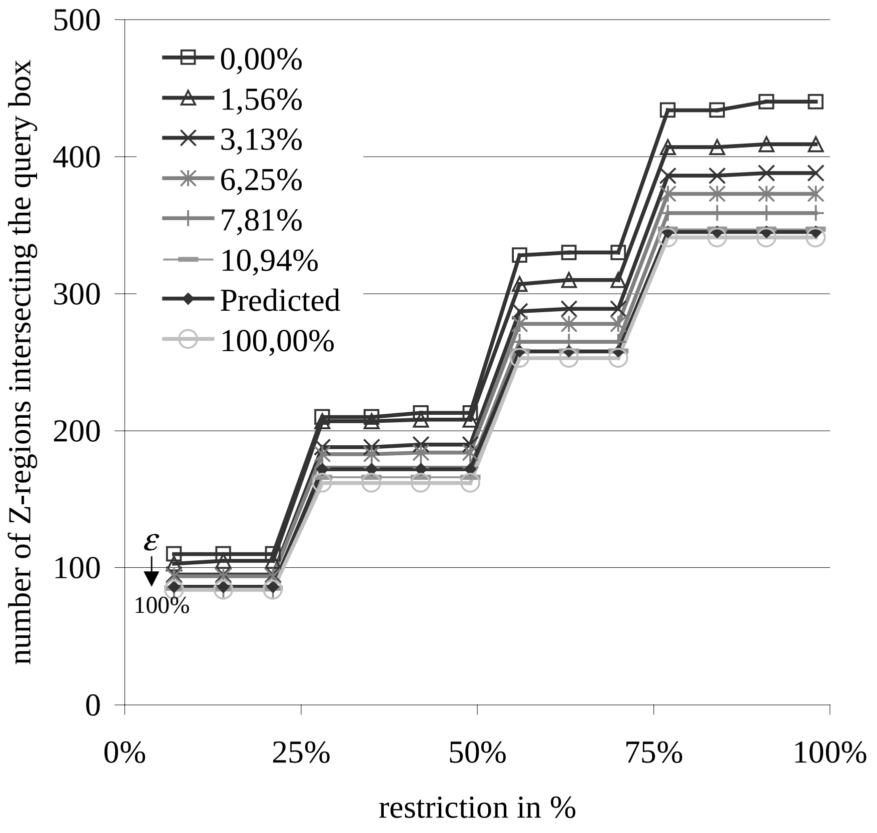


Figure 6-4: ε-split and cost function

6.2 A Cost Model

In the following we define a cost model to compare the cost of various access methods for range queries. Our cost model takes both CPU-time and I/O-time for query processing into account. For I/O-time we consider both clustered access and random access in our cost model.

Clustering places data that is likely to be accessed together physically close to each other. The goal of clustering is to limit the number of disk accesses required to process a query by increasing the likelihood that query results have already been cached. We distinguish two kinds of clustering: *Tuple clustering* stores tuples of one or several relations on one disk 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 disk in insertion order. *Page clustering* in addition to tuple clustering also maintains physical clustering between disk pages.

Let t_π be the (average case or worst case) positioning time of a hard disk, t_τ be the transfer time of a hard disk and t_ξ the CPU time[17] spent for page processing after retrieval. We assume that the prefetching strategy of the file system reads a physical cluster of L consecutive pages from disk with one random access into the read-ahead cache. This takes time $t_\pi + (t_\tau + t_\xi) \cdot L$. In addition we assume that each page has a capacity of C tuples. With this cost model page clustering (i.e., reading k tuples in consecutive order) takes time

$$c_{page}(C, L, t_\pi, t_{\tau,}, t_\xi ,k) = \min(\lceil \lceil k / C \rceil / L \rceil + 1, k) \cdot t_\pi + \max(\lceil k / C \rceil, L) \cdot (t_\tau + t_\xi)$$

Tuple clustering requires to position the read/write-head of the hard disk for every page and therefore takes time

$$c_{tuple}(C , t_\pi , t_\tau, t_\xi ,k)= \min(\lceil k/C+1 \rceil, k) \cdot (t_\pi + t_\xi + t_\tau)$$

Random access without caching requires to position the read/write-head of the hard disk for every tuple and therefore takes time

$$c_{random}(t_\pi, t_\tau, t_\xi ,k)= k \cdot (t_\pi + t_\tau + t_\xi)$$

In the following considerations we assume that a query retrieves large result sets. We consequently omit the "+1" in the cost formulas for page clustering and tuple clustering, since this additional summand is negligible for large result sets. Basically this means that we assume a best case distribution of the tuples on pages so that the first tuple of the result set also is the first tuple on a page (i.e., a result set of less than C tuples will always be stored on *one* page).

[17] Note that t_ξ heavily depends on the specific query. For complex restrictions like multidimensional intervals or IN-clauses of SQL t_ξ may be considerably higher than for simple single-attribute restrictions.

6.3 The Cost of Range Queries

For retrieving or sorting a relation in combination with multidimensional hierarchical restrictions we define cost functions for response times and intermediate temporary storage. Our analysis considers a UB-Tree, a composite secondary index (CSI, clustering B^*-Tree) over all attributes (foreign keys of each dimension and measure attributes) of a fact table, a single secondary index (SSI, non-clustering B^*-Tree) on the attribute with the least selectivity and a full table scan (FTS). In addition we analyze the performance of bitmap index intersection (BII), which combines the bitmaps of each restricted attribute to determine the result set of the query. Note that an index organized table (IOT) as introduced in section 2.3 can be regarded to be a special case of a CSI, which stores the entire tuple in the B-Tree leaf pages and therefore does not have to store any tuple identifier there.

An FTS to answer multidimensional range queries with selectivity s_j in dimension j can exploit prefetching techniques to reduce the number of random page accesses at the expense of having to read the entire table.

Using a CSI with a composite B^*-Tree in lexicographic order $A_1, ..., A_d$ allows one to use the index for the restriction in A_1 at the expense of having a random access for each page.

A SSI on A_j requires a random page access for each tuple satisfying the restriction in A_j, since no clustering of the tuples is available. The number of random accesses of an SSI is limited to P, if the row identifiers of the SSI are sorted and then processed in physical page order for data page retrieval. For point restrictions on the index attribute, sorting of row identifiers may even be avoided: index pages for tuples with identical index attributes may be organized in the physical order of the row identifiers. Then point restrictions will get a list of row identifiers sorted according to the physical location of the tuple. This makes a SSI not to degenerate and behave similarly to an FTS in worst case.

BII requires a random access for each tuple satisfying the restrictions in all attributes. In addition the corresponding part of each bitmap index has to be retrieved. In analogy to an SSI the result of BII is a bitmap which is used to access data pages in physical order. Thus multiple random accesses to one data page will not occur. Figure 6-5 shows how two specific bitmap indexes process a two dimensional partial match query.

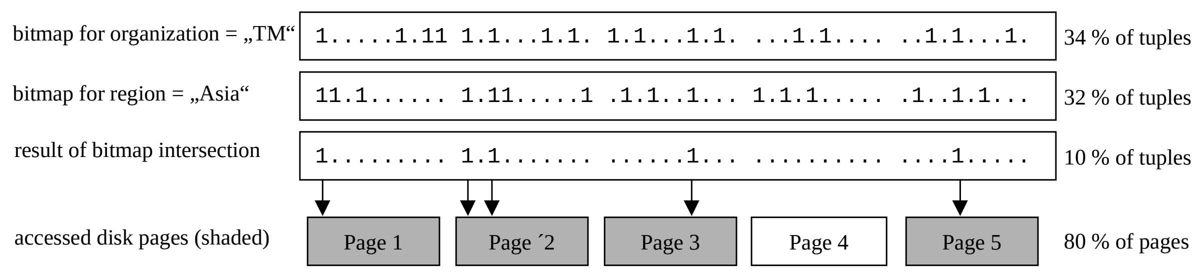


Figure 6-5 Bitmap index intersection

For each restriction, the bitmap is retrieved from the corresponding bitmap index. After intersecting these two bitmaps by a bitwise AND-operation the tuples corresponding to 1-bits are retrieved (The zero bits in the figure are denoted by a "." instead of a "0"). In the figure we assume C = 10 tuples to fit on one page, thus ten consecutive bits correspond to the tuples on one disk page. The selectivities for both dimensions are 32% respectively 34%, resulting in an overall selectivity of 10%. Since the data is not clustered on the pages, the query needs to retrieve 80% of the fact table to retrieve 10% of the tuples. In practice this ratio is even worse: Actual values for C range between 20 and 400 for 8kB pages.

The shaded part of each cube in Figure 6-6 shows the part of a three dimensional database which is retrieved by the corresponding access method to answer a three-dimensional range query with (s_1 = 25%, s_2 = 33%, and s_3 = 17%): an FTS retrieves the entire database exploiting clustering and prefetching. In contrast to that an SSI will rarely utilize any clustering benefits for small result sets. BII retrieves each bitmap by clustered access, whereas the data itself will often be spread over many data pages and then must be retrieved by random access to each page. However, for larger result sets the probability rises that prefetching might be applicable for bitmap indexes. This means, that BII will not be much less efficient than an FTS in worst case. A CSI with A_3 as first attribute in concatenation order utilizes clustering but only exploits the 17% restriction on A_3. In contrast to that the UB-Tree utilizes the restrictions of all dimensions and retrieves the data in a clustered way.

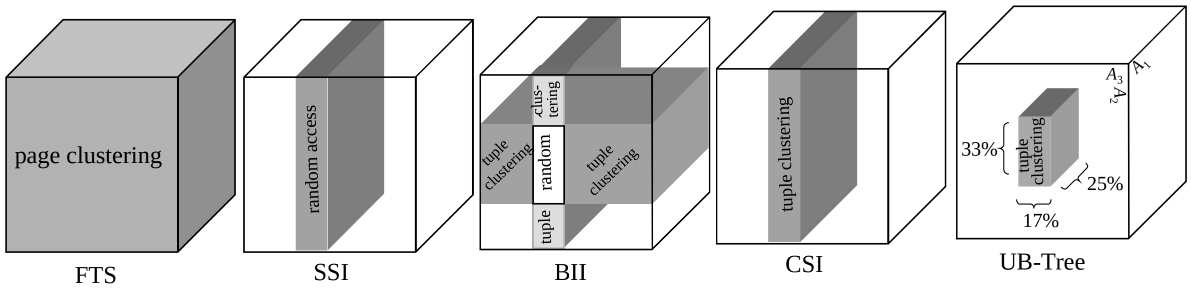


Figure 6-6: Access methods and clustering

6.3.1 Cost Functions

Using the cost model of Section 6.2 we calculate the cost of processing a fact table consisting of T tuples stored on P pages restricted by a multidimensional interval $Q = [[y, z]] = [y_1, z_1] \times \ldots \times [y_j, z_j] \times \ldots \times [y_d, z_d]$ with a selectivity of s_j in attribute A_j. For UB-Trees we assume a d-dimensional hierarchical organization of the table. For secondary indexes we assume B_j to be the size in pages of a secondary index on A_j. Figure 6-7 displays cost formulas for these access methods.

$$c_{\mathrm{FTS}}(d,P)=\left(t_\pi\cdot\frac{1}{L}+t_\tau+t_\xi\right)\cdot P$$

$$c_{\mathrm{CSI}}(d,P,s_1,\ldots,s_d)=\left(t_\pi+t_\tau+t_\xi\right)\cdot P\cdot s_1$$

$$c_{\mathrm{SSI\ on\ dimension}\ i}(d,T,s_1,\ldots,s_d,B_i)=\left(t_\pi+t_\tau+t_\xi\right)\cdot s_i\cdot B_i+\left(\frac{t_\pi}{\mathrm{prefetch}(T,P,s_1,\ldots,s_d)}+t_\tau+t_\xi\right)\cdot\frac{T\cdot s_i}{\mathrm{cluster}(T,P,s_1,\ldots,s_d)}$$

$$c_{\mathrm{BII}}(d,T,s_1,\ldots,s_d,B_1,\ldots,B_d)=\underbrace{\left(t_\pi+t_\tau+t_\xi\right)\cdot\sum_{i=1}^{d}\left(s_i\cdot B_i\right)}_{\text{bitmap retrieval \& intersection}}+\underbrace{\left(\frac{t_\pi}{\mathrm{prefetch}(T,P,s_1,\ldots,s_d)}+t_\tau+t_\xi\right)\cdot\frac{T\cdot\prod_{i=1}^{d}s_i}{\mathrm{cluster}(T,P,s_1,\ldots,s_d)}}_{\text{tuple retrieval by random access (prefetching for large result sets)}}$$

$$c_{\mathrm{UB}}(d,P,y,z)=\left(t_\pi+t_\tau+t_\xi\right)\cdot\prod_{j=1}^{d}n_j(d,P,y_j,z_j)=\left(t_\pi+t_\tau+t_\xi\right)\cdot P\cdot\prod_{j=1}^{d}\hat{s}_j$$

Figure 6-7: Cost functions for retrieval of a multidimensional interval

Interestingly, non-clustering indexes may utilize tuple clustering for large result sets, since the probability of tuples of the result set to be located on the same page increases with growing result set size. This “clustering effect” of non-clustering secondary indexes is taken into account by our cost functions by introducing a function *cluster* that with growing result set size of a query grows from 1 to *C* and interpolates linearly with the selectivity:

$$\mathrm{cluster}(T,P,s_1,\ldots,s_d)=\max(1,C\cdot\mathrm{selectivity}(Q))=\begin{cases}\max(1,C\cdot\prod_{i=1}^{d}s_i) & \text{for secondary index intersection}\\ \max(1,C\cdot s_1) & \text{for a single secondary index on } A_1\end{cases}$$

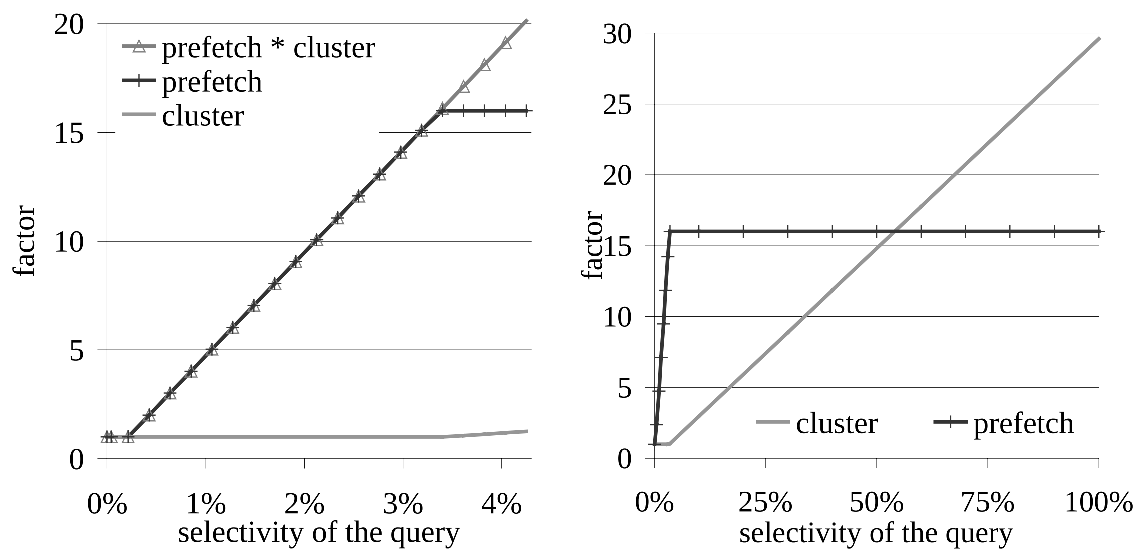
Thus min(cluster) = 1 and max(cluster) = *C*.

In the same way secondary indexes may utilize page clustering when retrieving large result sets. The function prefetch($T,P,s_1,\ldots,s_d$) denotes the actual prefetching for the retrieval of a result set by secondary indexes and grows from 1 to *L*: With growing result set size the probability of tuples to be located on consecutive pages increases.

$$\mathrm{prefetch}(T,P,s_1,\ldots,s_d)=\min\left(L,\max\left(1,\frac{T\cdot\mathrm{selectivity}(Q)}{P\cdot L}\right)\right)=\begin{cases}\min\left(L,\max\left(1,\frac{T\cdot\prod_{i=1}^{d}s_i}{P\cdot L}\right)\right) & \text{for SII}\\ \min\left(L,\max\left(1,\frac{T\cdot s_1}{P\cdot L}\right)\right) & \text{for SSI on } A_1\end{cases}$$

Thus min(prefetch) = 1 and max(prefetch) = *L*.

For T = $26\cdot10^6$ tuples on P = $878\cdot10^3$ pages, *C* = 30 and *L* = 16, Figure 6-8 shows the functions prefetch and cluster depending on the selectivity of the query. As soon as the selectivity exceeds 1/*C* (=3,33%) several tuples of the result set are stored on one page. Prefetching already shows benefits when the selectivity exceeds 1/(*L*·*C*) = 0,21%.



(a) (b)

Figure 6-8: prefetch(T, P, s_1, ..., s_d) and cluster(T, P, s_1, ..., s_d)

These definitions are used for prefetch and cluster in the cost functions of Figure 6-7.

Note that in order to utilize tuple clustering or page clustering the tuple identifiers of a secondary index (intersection) need to be sorted with respect to the physical storage order. Sorting can be avoided, if the restriction is a point restriction and tuples with identical index keys are stored in physical order in the secondary index. Note that tuples are delivered in physical order and not in the sort order of any index, if clustering is utilized in this way by secondary indexes. With optimized storage structures, like bitmap representation for row identifiers, partial match queries can be handled efficiently. However, the loss of the index key order limits the usability of the approach for range queries in combination with sort operations on an index attribute (see Section 6.4).

6.3.2 Simulation Results

Current operating systems usually prefetch L = 16 pages with one random access. We assume t_π = 10 [ms], t_τ = 0,6 [ms] and t_ξ = 0,4 [ms]. We assume a four dimensional organization of the UB-Tree.

Figure 6-9a shows the cost [in s] for a range query with (s_1 = 33%, s_2 = 25%, s_3 = 17%, s_4 = 100%) against 4-dimensional UB-Trees compared to other access techniques. The table size is varied from one page up to one million pages.

Varying the selectivity of the restriction in A_1 for a table size of P = 878k pages (about 7GB for a page size of 8kB) shows that UB-Trees are superior to a CSI on the most selective attribute A_3, since this CSI cannot exploit any restriction but the one on A_3. UB-Trees can exploit the restrictions on A_1 and A_2 in addition to the restriction on A_3. Thus UB-Trees are also superior to an FTS and to BII using bitmap indexes for all four dimensions (Figure 6-9b).

Note that due to the high selectivity in A_3 a SSI always degenerates to an FTS. For an overall selectivity of 75% · 17% · 25% · 100% = 3,1875 % an FTS is already preferable to BII. Since bitmap indexes do not cluster the data, the result set defined by the restrictions in all dimensions must be sufficiently small for BII to be competitive.

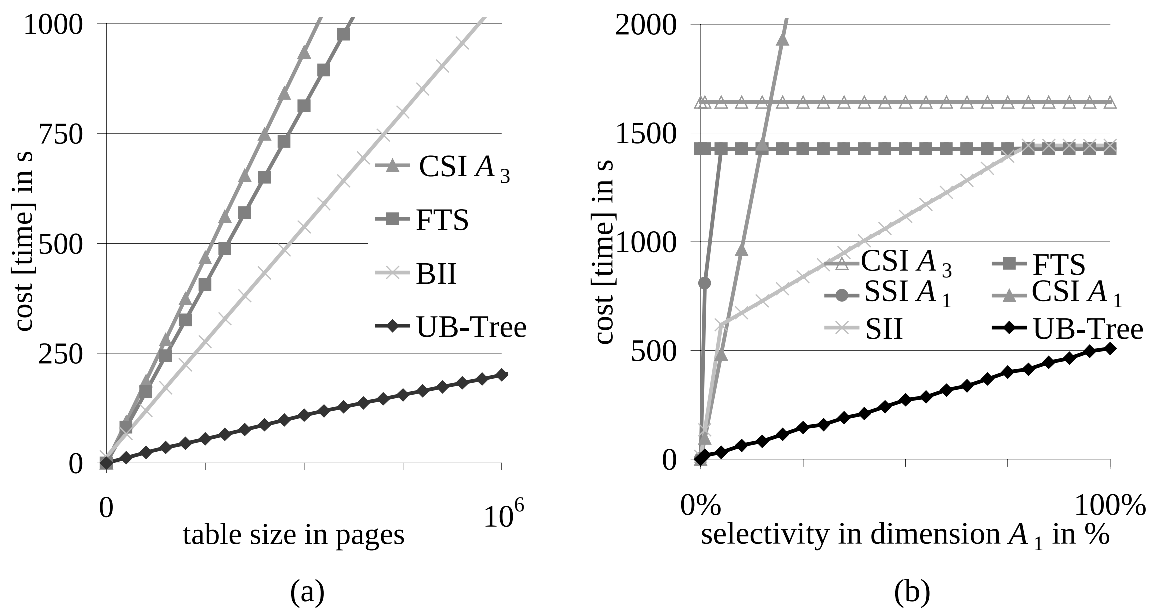


Figure 6-9: Simulation of a four dimensional range query

6.4 The Cost of Range Queries with Sort Operations

The Tetris Algorithm of Section 4.7 allows one to process queries with multi-attribute restrictions and sort operations. Since only very little main memory is needed, in general no external sorting will be necessary. In this section the response times and storage requirements of the Tetris algorithm are compared to those of a merge sort algorithm in combination with any of the access methods of Section 6.3.

6.4.1 Cost Functions for Secondary Storage

The Tetris cache is considerably smaller than the temporary storage of $P \cdot \Pi s_i$ required by the merge-sort algorithm that is necessary after the data has been retrieved by an FTS or any index on a restricted attribute. To sort A_j the Tetris algorithm just requires to cache one slice, i.e.,

$$cache_{\text{Tetris}}(d, P, y, z, j) = \prod_{\substack{i=1,\dots,d \\ i \neq j}} n_i(d, P, y_i, z_i)$$

For a two-dimensional UB-Tree the above formula results in a square root function of the number of Z-regions overlapping the query box, i.e., $cache_{\text{Tetris}}(2, P, s_1, s_2, j) \approx \sqrt{P \cdot s_1 \cdot s_2}$ for $j \in \{1, 2\}$.

6.4.2 Cost Functions for Response Time

For the following considerations we assume a merge sort algorithm using a main memory of M pages and a merge degree of m. We divide the sort process in a retrieval phase (which retrieves the data to create initial runs for the merge-sort) and a sort phase (which actually performs the merge-sort). Because of the multi-attribute filtering of the retrieval phase the data set to be sorted is usually smaller than the entire table. With s_j denoting the selectivity of the restriction in attribute A_j and independence of the attributes, $P \cdot \Pi s_i$ pages need to be sorted. The cost functions of Section 6.3 can be used to calculate the cost of the retrieval phase to create the initial runs for the sort operation. If an access method does not return the tuples in the requested sort order, sorting with the cost of c_{sort} takes place. If $M \geq P \cdot \Pi s_i$, sorting takes place in main memory. c_{sort} then is the cost of an internal sort operation. c_{sort} is zero, if a CSI with A_1 as first attribute is used for the retrieval phase and the sorting attribute is A_1, since the data then is already retrieved in the desired sort order.

$$c_{\text{sort}}(d,P,C,m,M,s_1,\ldots,s_d) = \begin{cases} t_\xi \cdot P \cdot \prod_{i=1}^{d} s_i \cdot \log\left(P \cdot \prod_{i=1}^{d} s_i\right) & \text{, if } M > P \cdot \prod_{i=1}^{d} s_i \\ \underbrace{\left(t_\pi \cdot \frac{1}{C} + t_\tau + t_\xi\right)}_{\text{consecutive access}} \cdot \underbrace{2}_{\substack{\text{read \&} \\ \text{write}}} \cdot \underbrace{\left(P \cdot \prod_{i=1}^{d} s_i\right)}_{\text{pages to sort}} \cdot \underbrace{\log_m\left(\frac{P}{M} \cdot \prod_{i=1}^{d} s_i\right)}_{\text{number of merge phases}} & \text{, otherwise} \end{cases}$$

As shown in the section before, SSI and CSI on restricted attributes are only efficient for fairly small result sets. In this case sorting would take place in main memory. One can expect that as soon as external sorting is necessary, SSI and CSI are not efficient for the retrieval phase anymore. Therefore we do not consider SSI and CSI in this section, since we focus on external sorting here.

The Tetris algorithm has to sort each cached slice. Since the algorithm reads $n(y_j, z_j, l_j)$ slices, the overall cost of internal sorting according to A_j is:

$$c_{\text{Tetris}}(d,P,y,z,j) = t_\xi \cdot n(y_j, z_j, l_j) \cdot cache_{\text{Tetris}}(d,P,y,z,j) \cdot \log cache_{\text{Tetris}}(d,P,y,z,j)$$

6.4.3 Cost Functions for Interactive Response Times

When the Tetris algorithm has completed a slice, it is usually sorted internally and then is available in sort order. Thus first results are available for further processing after a time of $cache_{\text{tetris}}(d,p,x,y) \cdot (t_\pi + t_\tau + t_\xi)$. For a CSI (or IOT) on a restricted attribute and an FTS it is necessary to wait until the entire merge sort is completed. This yields a tremendous performance advantage of the Tetris algorithm for pipelined processing and interactive response times.

6.4.4 Simulation Results

Using the same parameters as in Section 6.3 and additionally using a main memory cache of 32 MB and a merge degree of $m = 2$ for the merge sort algorithm Figure 6-10 shows the cost [in s] for sorting the result set of a fact table defined by restrictions in multiple hierarchical dimensions

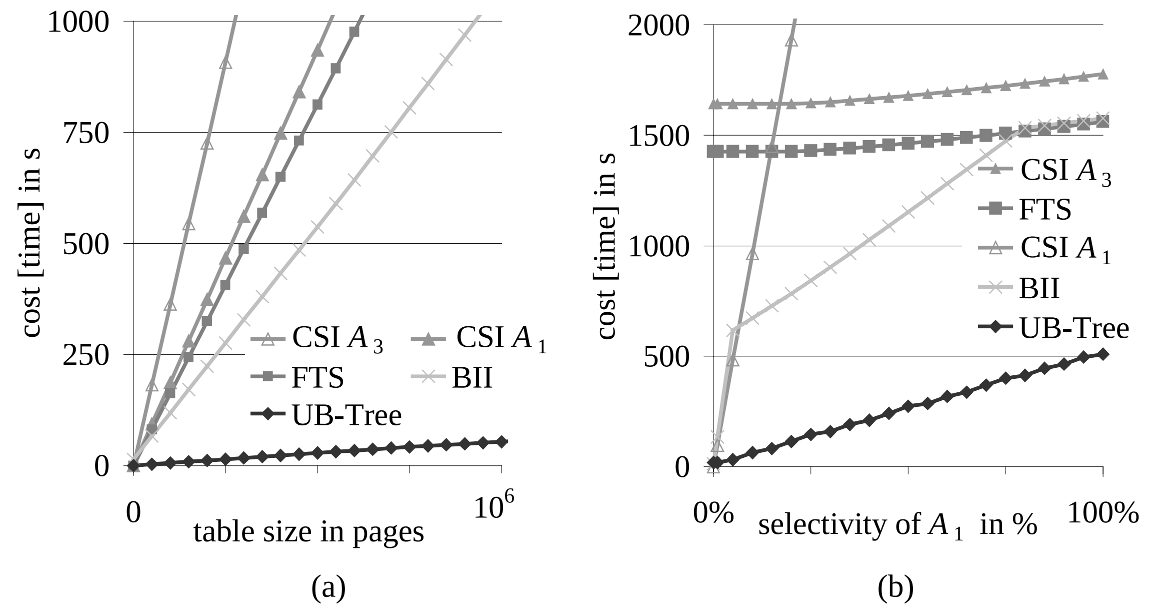


Figure 6-10: Simulation of sorting a four dimensional query box

Figure 6-10a shows the cost [in s] for sorting a four dimensional query box with (s_1 = 33%, s_2 = 25%, s_3 = 17%, s_4 = 100%) according to A_1. Again the table size is varied from one page to one million pages. Since the selectivity of each restriction exceeds 10%, processing this query by a single secondary index usually degenerates to an FTS. The speed up of the Tetris algorithm for UB-Trees grows superlinearly with increasing table size, since all other access methods require an external merge sort. Varying the selectivity of the restriction in A_1 for a table size of P = 878k pages in Figure 6-10b shows the superiority of the Tetris algorithm, since multidimensional clustering allows one to exploit multi-attribute restrictions to reduce the number of random accesses and at the same time avoids an expensive external sort operation. Sorting with Tetris takes place in main memory as long as this memory suffices to hold one slice of the query box. Figure 6-11(a, b) shows that the temporary storage for the merge sort algorithm used by FTS, BII, and CSI A_3 soon exceeds the main memory sorter cache of M = 32 MB when processing the queries of Figure 6-10 (a, b). In contrast to that sorting with Tetris never requires more than 14 MB of cache for one slice and thus sorting can take place in main memory.

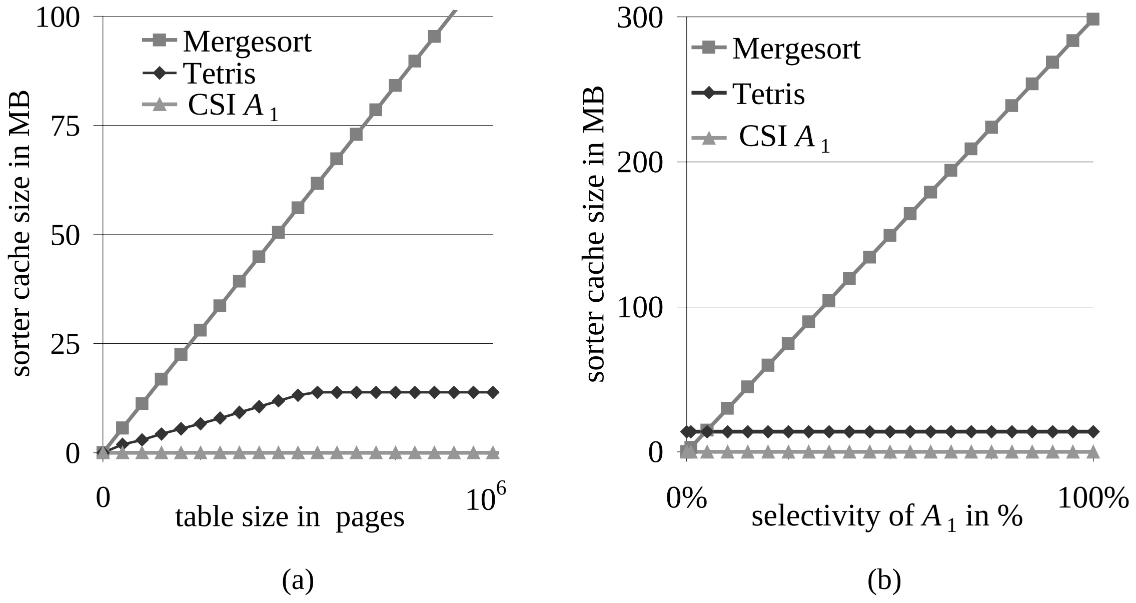


Figure 6-11: Temporary storage required for the sorter cache (simulation)

A CSI or SSI on A_1 does not require any sorter cache. The tradeoff of these two access methods is the inability to use restrictions in multiple dimensions. Overall, the Tetris algorithm for UB-Trees outperforms any access method either with respect to response time or with respect to both response time and temporary storage requirements.

6.5 Summary: Cost Analysis

Using our cost functions we found out that for sort operations with restrictions in some attributes UB-Trees and the Tetris algorithm are superior to one-dimensional access methods, unless a strongly preferred sort order on one attribute per relation exists or the restrictions are not selective enough to make up the tenfold speed of the FTS. Another limitation of our technique is the number of dimensions as investigated in Section 3.9: Increasing dimensionality exponentially reduces the potential of the multidimensional space partitioning to create a total sort order in one dimension. Our theoretical and practical analysis shows that multidimensional indexes of up to 6 dimensions are handled very well with table sizes around 1 GB. These dimensionalities are typical for data warehousing applications and in particular for the TPC-D benchmark (see Section 8.1). With larger table sizes even further attributes could be added to the multidimensional index in order to speed up queries with restrictions or sorted processing in this attribute.

In science, as in life, learning and knowledge are distinct, and the study of things, and not of books, is the source of the latter.

(Thomas H. Huxley)

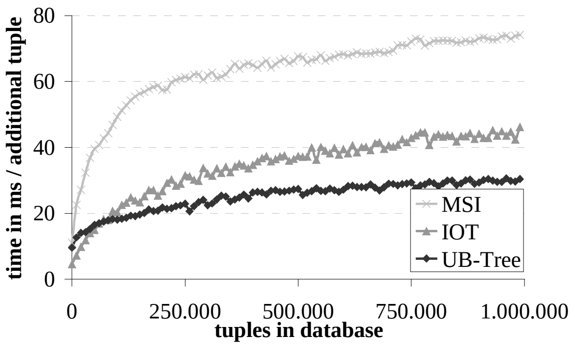
Chapter 7

Performance Measurements

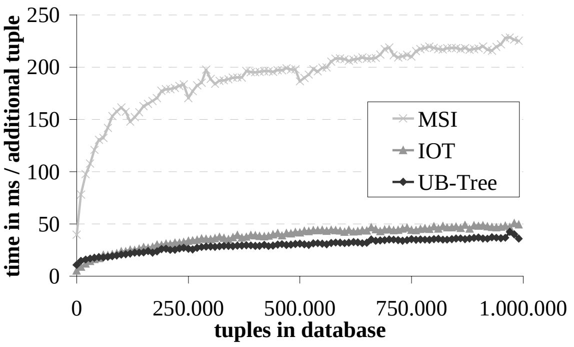
Large-scale experiments with our prototype implementation are reported in this chapter. We investigate the behavior of our prototype implementation of the UB-Tree for relations storing artificially generated, uniformly distributed data of up to 10 million tuples. We show the performance of insertion into UB-Trees, clustering B-Trees and multiple non-clustering B-Trees for various dimensionalities. After a brief summary of main results of point query investigations we more closely look on the range query performance. We first identify the main factors that influence the range query behavior of the UB-Tree. Then several types of measurements are defined. Performance figures for range queries measured with our prototype implementation on top of two relational DBMS are listed and interpreted. Due to legal considerations the DBMS have been made anonymous. Last but not least performance figures are given for partial match queries that restrict only a subset of the dimensions of the UB-Tree. This chapter practically undermines our observations from Chapter 3 and also proves the accuracy of our cost functions from Chapter 6 for uniformly distributed data.

7.1 Insert Performance

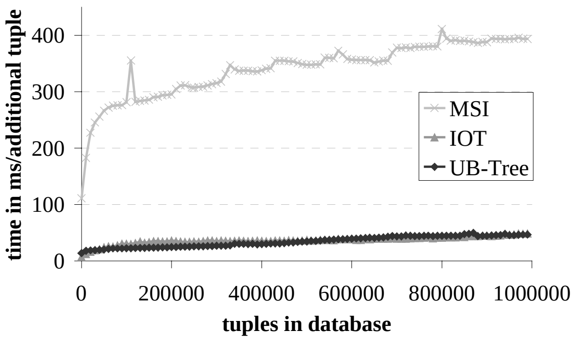
Figure 7-1 shows performance measurements of the insert performance of the prototype implementation on DBMS1 on a 167 MHz ULTRA SPARC 2 with an IBM hard disk with 8ms positioning time for 3-dimensional (a), 6-dimensional (b), and 12-dimensional data (c). Each measurements series shows the time in ms for the insertion of one additional tuple for the database size shown on the horizontal axis. The performance of UB-Trees is compared to insertion into an index organized table (IOT) on the attributes and to insertions into multiple secondary indexes over 3, 6 resp. 12 dimensions (MSI).



(a)



(b)



(c)

Figure 7-1: Insert performance

Figure 7-2 shows how the insertion time for UB-Tree insert is distributed to CPU for address calculation, I/O for page retrieval, CPU for page modification and I/O for page update. The time distribution in the bars is from top to bottom as listed in the legend.

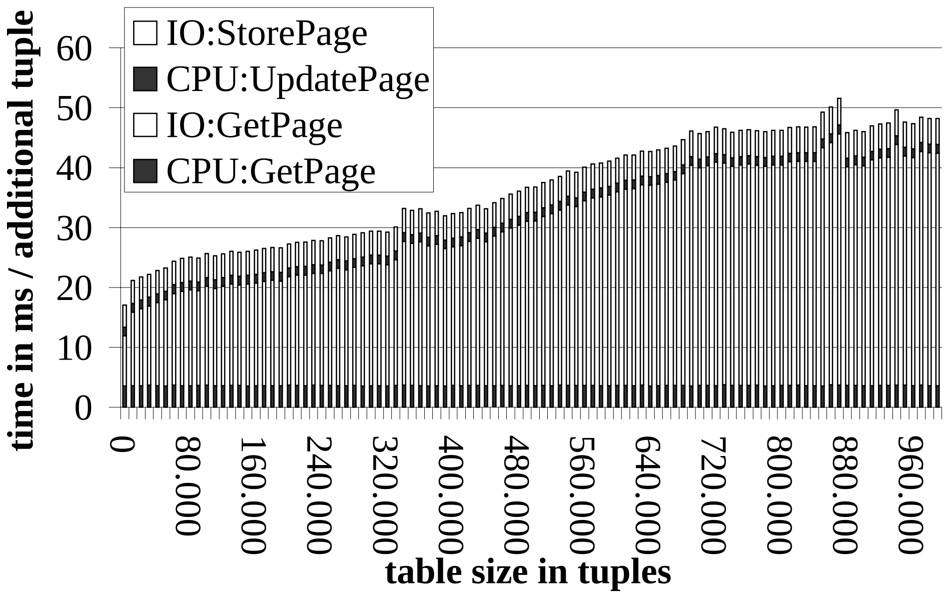


Figure 7-2: Time distribution for UB-Tree insertion

All in all the performance of UB-Tree insertion is similar to insertion into an index organized table. This is not surprising, since the UB-Tree merely requires CPU operations for address calculation in addition to index organized tables. This additional address calculation overhead is negligible. For a 6-dimensional integer tuple it uses less than 1% of the total insertion time. For a UB-Tree of height h UB-Tree insertion is about $(h-1)/h \cdot d$ times faster than insertion into multiple B-Trees. The factor $(h-1)/h$ in the above formula is due to the fact that for a given database the UB-Tree in the average is one level higher than each of the secondary indexes.

Detailed measurements and investigations of insert performance on the prototype implementation of UB-Trees can be found in [Fri97] and [Bau97].

7.2 Exact Match Query Performance

A point can be found in $O(\log_k T)$ time, where T is the number of objects in the relation and $k = ½C$, since UB-trees are balanced and searched exactly like the variant of B-tree used as the underlying data structure for the UB-tree. Thus the point query performance of a UB-Tree is similar to that of an IOT. The additional address calculation overhead is negligible. Detailed measurements and investigations of exact match query performance on the prototype implementation of UB-Trees on DBMS1 can be found in [Fri97].

7.3 Storage Requirements

As expected, the storage requirements of UB-Trees are equivalent to those of concatenated B-Trees (IOTs). UB-Trees require more storage than a simple sequential file organization (FTS), since one additional B-Tree is maintained in order to store the Z-addresses of the multidimensional space partitioning. However, this yields a tremendous reduction in storage requirements over multiple secondary indexes (MSI), which require to maintain one B-Tree for each dimension.

	FTS	IOT (A_1..A_6)	MSI (6 dim.)	UB-Tree
Table Size in Pages / MB	21115 / 42,2	27759 / 55,5	76678 / 153,3	21616 / 43,2
Height of B-Tree	-	4	3	4
Leaf Page Utilization	100%	89,677%	90,003%	99,301%

Table 7-1: Table Sizes of a 1 million 28 Byte tuples relation on DBMS1 (2kB pages)

Table 7-1 and Table 7-2 show the storage requirements for a one million respectively ten million tuple DBMS1 relation organized as FTS, IOT, 6 multiple secondary indexes and a 6-dimensional UB-Tree on 2kB pages. Due to the simpler page handling of our prototype implementation (e.g., no variable length attributes, only support for numbers and character data), storage requirements for UB-Trees are lower than those of native DBMS1 B-Trees.

	FTS	IOT (A_1..A_6)	MSI (6 dim.)	UB-Tree
Table Size in Pages / MB	256082 / 512,2	296284 / 592,6	775951 / 1551,8	211218 / 422,4
Height of B-Tree	-	4	4	5
Leaf Page Utilization	100%	89,677%	90,003%	99,301%

Table 7-2: Table Sizes of a 10 million 28 Byte tuples relation on DBMS1 (2kB pages)

For DBMS2 Table 7-3 shows the storage requirements for several 6-dimensional relations with different tuple sizes and table sizes. Due to some implementation overhead of B-Trees in DBMS2, the table size of a sequential file (FTS) for small relations is considerably lower than the table size of B-Trees as used for IOT and UB-Tree.

Tuples	125.000	250.000	1 million	1 million	2 million	4 million
Tuple Size	428 Byte	428 Byte	28 Byte	228 Byte	228 Byte	228 Byte
FTS Pages / MB	7814 / 61,0	15629 / 122,1	6274 / 49,0	34484 / 269,4	68969 / 538,8	137934 / 1077,6
IOT Pages / MB	13915 / 108,7	27810 / 217,3	11700 / 93,5	60705 / 474,3	96124 / 947,6	153420 / 1198,6
UB-Tree Pages / MB	13629 / 106,0	27194 / 212,5	5059 / 39,5	48119 / 376,0	121295 / 751,0	166669 / 1302,1

Table 7-3: Table Sizes of several tables on DBMS2 (8kB pages)

7.4 Range Query Performance

Answering a range query over a database, which is organized as a UB-tree, requires time proportional to the number of Z-regions overlapping the query box. For non-uniformly distributed data this number is not just a function of the restriction in each dimension, but also depends on the data distribution. Thus the parameters that influence the behavior of the UB-Tree range query algorithm are:

- query box volume
- query box position
- query box width in each dimension (degree of query box generation [Fri97])
- table size
- Z-region partitioning (i.e., data distribution, split parameters, etc.)
- dimensionality

Many of these parameters were investigated in detail in the master theses of Nils Frielinghaus [Fri97] and Roland Pieringer [Pie98]. There the cost function of Section 6.1 was used to simulate idealized uniformly partitioned UB-Trees. In addition measurements on a prototype implementation of the UB-Tree were conducted for various dimensionalities, database sizes, and data distributions. Here we just sketch the main results and present the most interesting measurement series.

For uniformly distributed data the range query performance exponentially decreases with the dimensionality of the UB-Tree (cf. also Section 3.9). The more dimensions are restricted, the better the range query performance gets, since each restriction is utilized by the UB-Tree (see also Section 7.4.3). However, restricting a dimension to less than 2^{-l} of its domain for a split level of l in that dimension does not further reduce the number of pages, since the split level defines the limit of resolution of the partitioning grid. The position of the query box on the partitioning grid can increase or decrease the number of regions overlapped by the query box even for uniformly distributed data and therefore influences the range query performance [Fri97]. The finer the grid, the less this effect may be observed. Since the grid is finer for larger databases or lower dimensionalities, dimensions that are not restricted by a query harm the query performance: An entire slice of Z-regions with respect to the not restricted dimension has to be retrieved. Summing up, the dimensions stored in the UB-Tree should be used for restriction. Thus proper index modeling is still necessary with UB-Trees, since the curse of high dimensionality forbids to add all attributes as dimensions to an index.

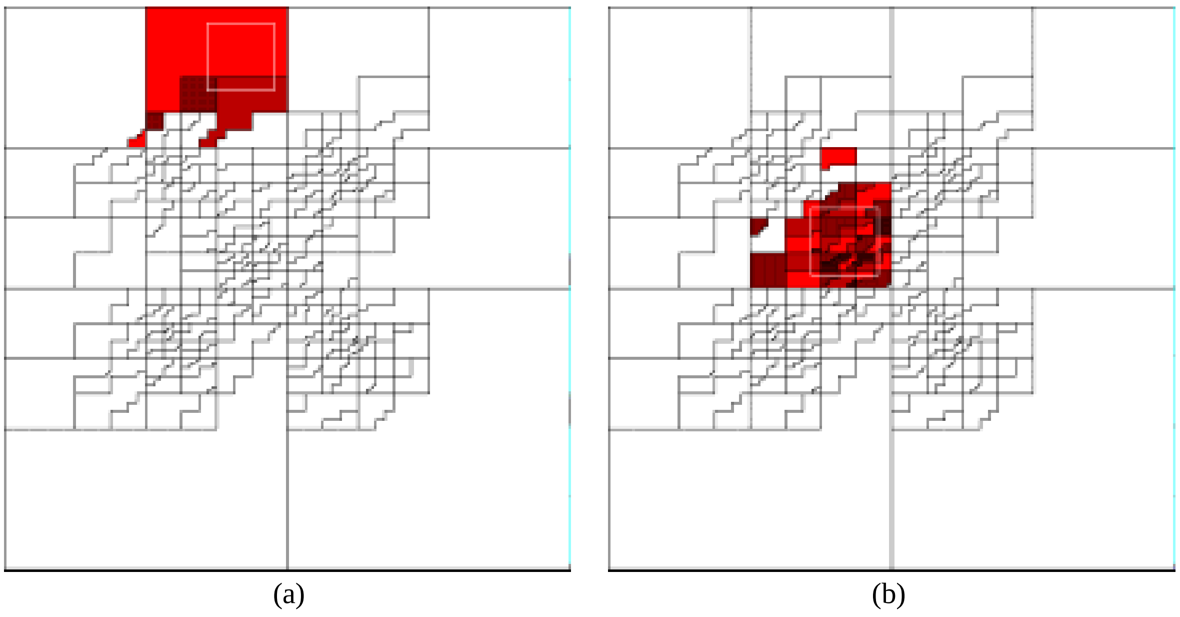


Figure 7-3: Range queries in sparsely (a) and densely (b) populated parts of a universe

In Figure 7-3 (a and b) range queries against the same non-uniformly distributed UB-Tree are shown. In this Figure the Z-regions are shaded that intersect the query box. The query box of Figure 7-3b has a result set of 617 points and overlaps 27 regions. Although the query box of Figure 7-3a has the same volume, it only covers a sparsely populated part of the universe and thus only 78 points in 3 regions are retrieved by the range query. Since a Z-region corresponds to a leaf page of the UB-Tree, Figure 7-3 shows that the number of disk accesses is proportional to the result set size of the range query.

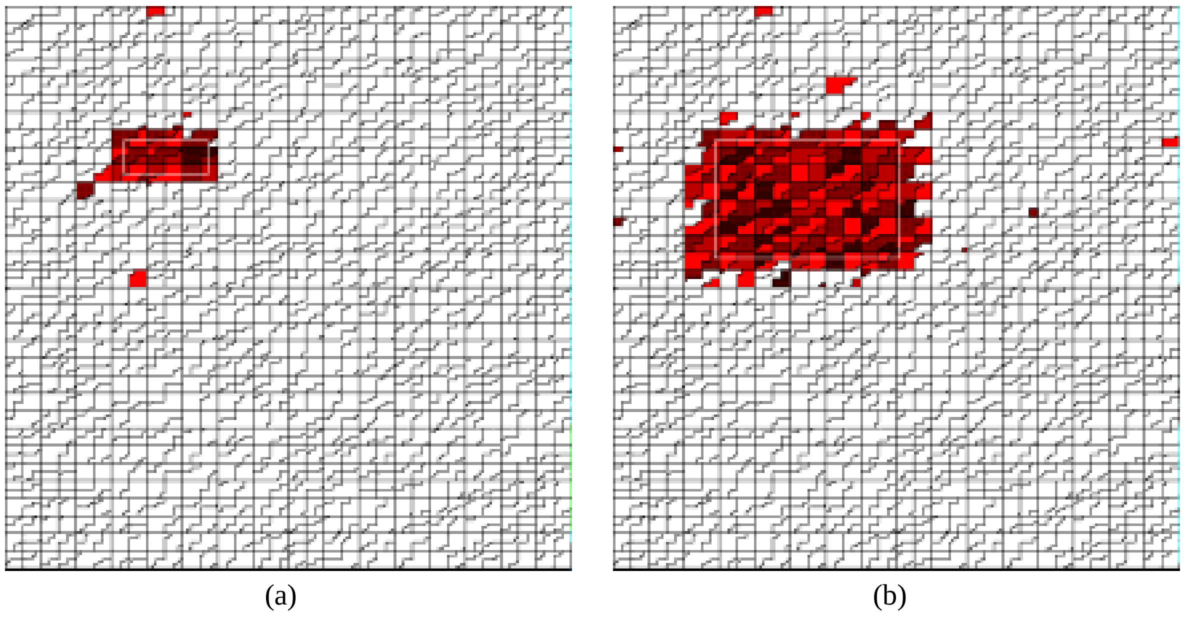


Figure 7-4: Query box volumes

Figure 7-4 (a and b) show two query boxes of different volume located in different parts of a uniformly distributed UB-Tree. The larger the query box in volume, the more Z-regions are overlapped by the query box and retrieved by the range query algorithm. Since with a larger query box volume many Z-regions are entirely contained in the query box, one can also state that the number of retrieved Z-regions is proportional to the result set of the query box.

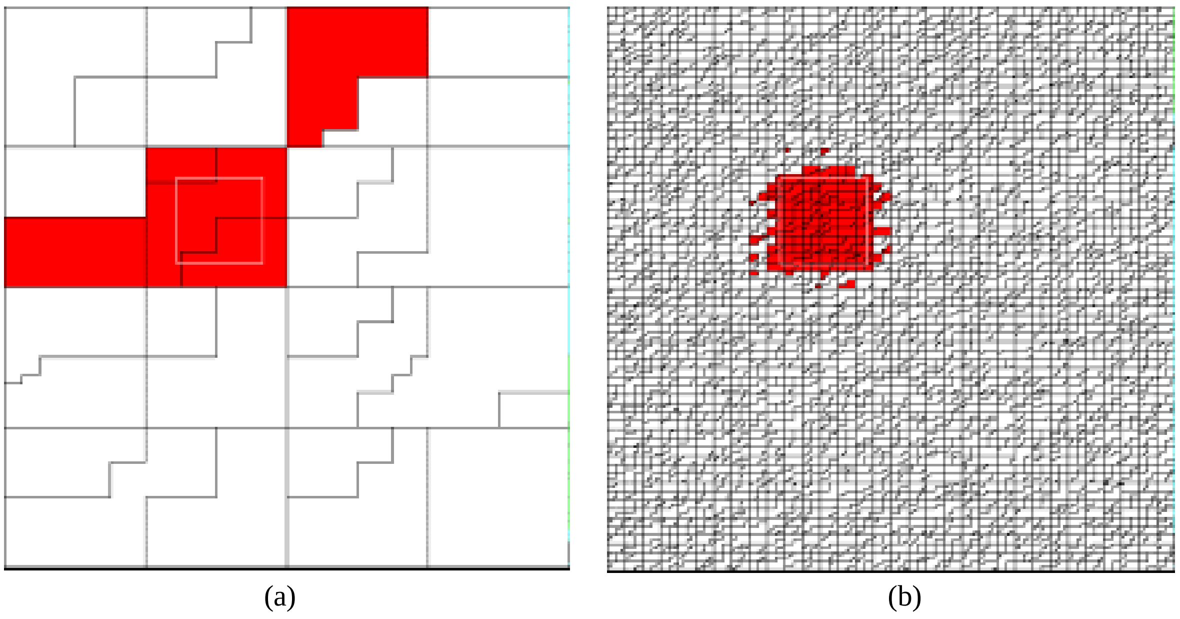


Figure 7-5: Range queries and scalability

Figure 7-5a displays a range query against a UB-Tree storing 1000 tuples on 25 Z-regions. The UB-Tree in Figure 7-5b stores 50000 tuples on about 2500 Z-regions. Thus Figure 7-5 shows that the query box is approximated more closely by the Z-region partitioning as the database increases.

7.4.1 Types of Measurements

Definition 7-1 (range method measurement for an access method): A *range query measurement for an access method* is an experiment that executes a range query on a single table with ranges in *d* attributes $A_1,...,A_d$ using a certain access method for table access and measures certain parameters (e.g., response time or number of I/Os) of the query execution. Depending on the access method used for the table access we speak of

- *FTS measurement* for full table scans
- *IOT A_i measurement* for an index organized table on attribute A_i as first key in concatenation order
- *UB-Tree measurement* for UB-Trees
- *SSI A_i* measurement for single secondary index on attribute A_i
- *SII measurement* for intersection of secondary indexes

Definition 7-2 (measurement for a set of access methods): A *measurement* uses the same restrictions to perform a range query measurement for each access method of a set of access methods.

Definition 7-3 (measurement series): A *measurement series* is an ordered sequence of measurements for a set of access methods.

In the following sections we investigate two types of measurement series on a relation with *d* index attributes, namely *c*%-measurement series and cube measurements series.

Definition 7-4 (*c*%-measurement series): A *c%-measurement series* restricts *d*-1 attributes to *c*% of their domain, whereas one attribute (the variable attribute) is varied from 0% to 100%.

For independently uniformly distributed data the selectivity of a *c*%-measurement with a selectivity of *x*% in the variable attribute is $c\%^{d-1} \cdot x\%$.

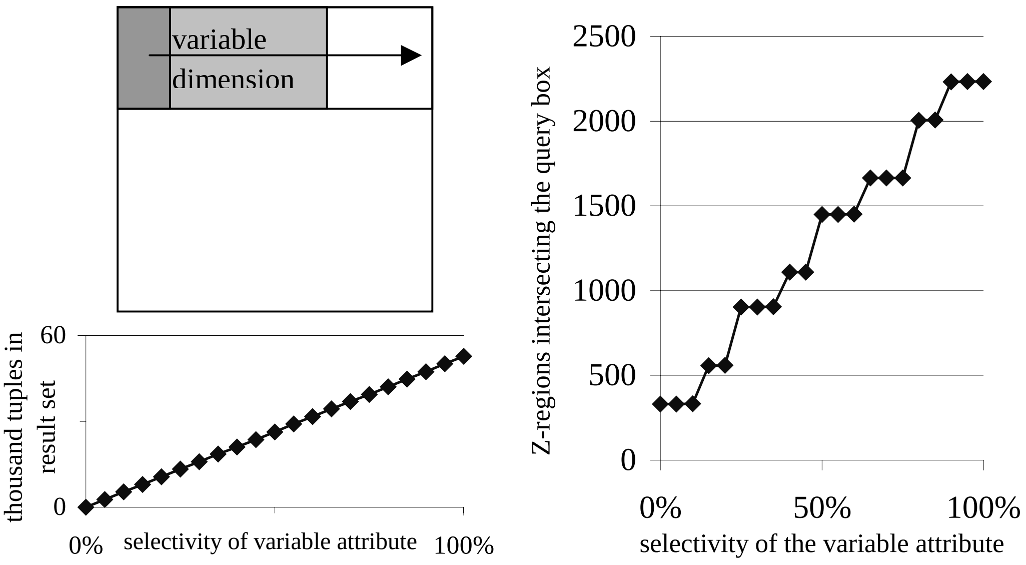


Figure 7-6: Characteristics of a c%-measurement series

The upper left part of Figure 7-6 shows the volume of the query box for three measurements of a two-dimensional *c*%-measurement series. Below that two-dimensional visualization the Figure shows the linearly growing result set size for a 35%-measurement series against a six-dimensional database storing 10^7 tuples. The right part of Figure 7-6 shows the number of Z-regions intersected by the query boxes of a 35%-measurement series against a six-dimensional UB-Tree storing 10^7 uniformly distributed tuples on 211218 Z-regions. In the Figure one can see a typical characteristic of *c*%-measurements for UB-Trees: The number of Z-regions intersecting the query box is a staircase function for a *c*%-measurement series. This staircase behavior is due to the fact, that for uniformly distributed data the Z-region partitioning is a discrete *d*-dimensional grid with clearly defined partitioning points (which are the points 50%, then 1/4 and 3/4, then $1/2^3$, $3/2^3$, $5/2^3$ and $7/2^3$, etc. depending on the data base size, cf. Section 6.1). A small enlargement of the query box in the variable dimension does not exceed a grid point and thus does not cause any further Z-region to be intersected. As soon as the query box exceeds a grid point, one additional slice of Z-regions is intersected. The height of the staircase (i.e., the number of additionally intersected Z-regions) in the new slice of the grid depends on *c*%, the restriction in the variable dimension and on the number of dimensions of the grid. A larger value for *c*% means that the query box covers a larger part of the multidimensional space and therefore a higher number of Z-regions will be intersected when exceeding a grid point. Similarly, with a higher dimensionality more Z-regions are intersected when partitioning point is exceeded. With a linearly growing result set the staircase function approximates a linear function. Further analysis of the staircase phenomenon can be found in [Fri97].

Definition 7-5 (cube measurement series): A *cube measurement series* varies the restriction on all attributes from 0% to 100% at the same time.

For a cube measurement on independently uniformly distributed data the selectivity of the restriction in each dimension is identical. Thus a selectivity of *x*% in each dimension results in an overall selectivity of $x\%^d$ for the query. In analogy to Figure 7-6, Figure 7-7 shows the volumes for three measurements of a two dimensional cube measurement series. In addition it shows the result set size and the number of intersected Z-regions for a cube measurement series against a 6-dimensional UB-Tree of 10^7 tuples on 211218 pages. The number of intersected Z-regions again shows a staircase behavior, which in this case approximates the polynomial function.

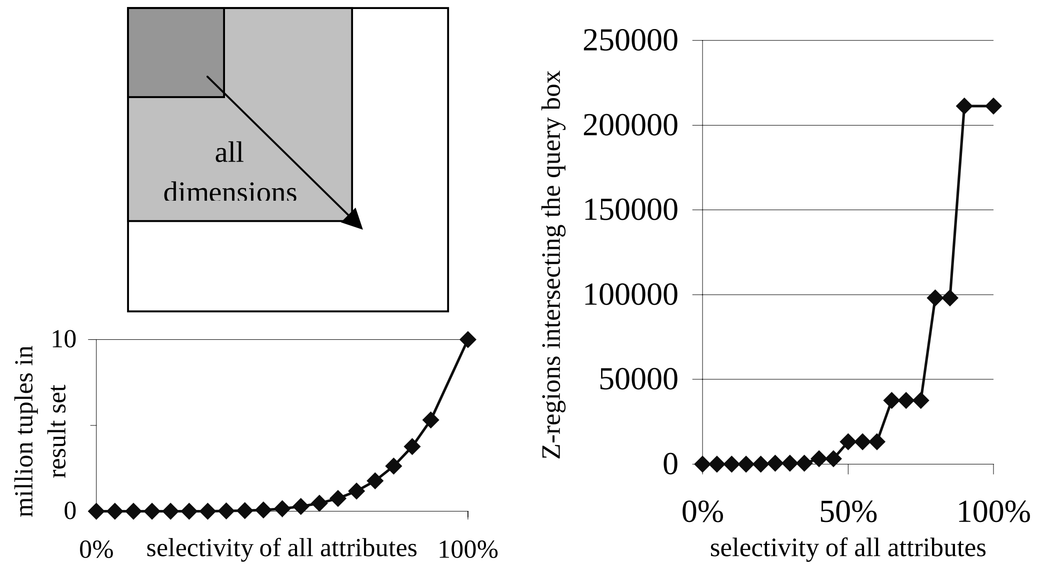


Figure 7-7: Characteristics of a cube measurement series

Thus with growing restriction in the variable attribute(s) the result set of a cube measurement series grows polynomially, whereas the result set of a *c*% measurement series grows linearly. This means, that cube measurement series are a good way to theoretically analyze the performance degeneration of a multidimensional index from very small to very large result sets ranging from $(1\%)^d$ (is usually just one or very few tuples) to 100% (the entire relation). *c*% measurement series indicate whether an index degenerates if the restriction is varied only in one attribute and therefore allow to judge the symmetry of a multidimensional index over the dimensions.

7.4.2 Comparative Performance Measurements

With the prototype implementation of the UB-Tree performance measurements were conducted on several commercial DBMS. Here we just list the results of DBMS1 and DBMS2. The results on the other DBMS are qualitatively equivalent to either DBMS1 or DBMS2 and can be found in [Ova99] and [Pfa99]. To get comparable results, precautions were taken in order to eliminate caching effects. Descriptions of these precautions can be found in [Fri97] for DBMS1 and [Pie98] for DBMS2.

Because of different resources and configurations of the database servers for DBMS2 and DBMS1, the measurements on both DBMS are not comparable quantitatively. This holds especially because of the different table sizes. However, a qualitative comparison is possible. Qualitatively, the performance gain of the UB-Tree compared to an IOT is identical for DBMS2 and DBMS1. The only qualitative difference is the performance of an FTS: DBMS1 implements relations as IOTs; an FTS retrieves the data in primary key order by tuple clustered access. Thus an FTS in DBMS1 is identical to reading 100% of a relation via an IOT. In contrast to that DBMS2 utilizes page clustering for FTSs. Therefore an FTS in DBMS2 is more efficient than in DBMS1.

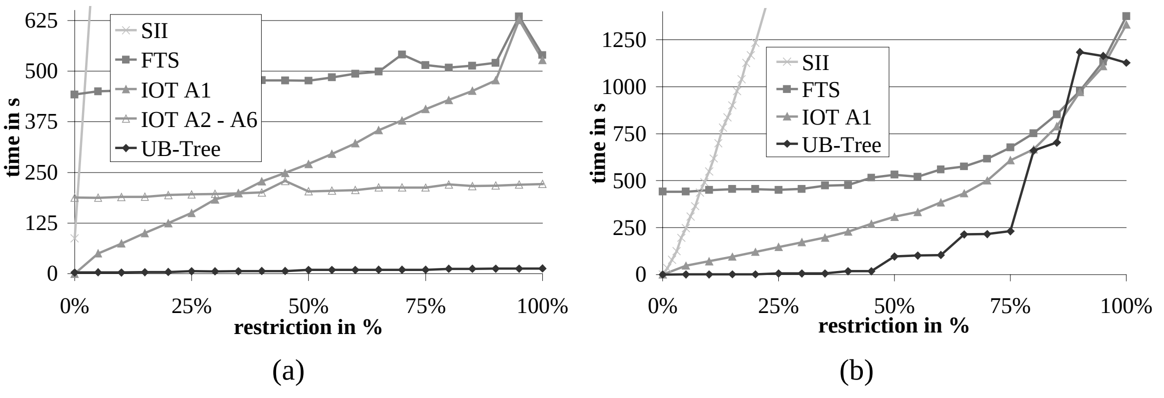


Figure 7-8: DBMS1

The measurements of Figure 7-8 were conducted on DBMS1 on a SUN ULTRA SPARC II 167 MHz with an IBM 8ms hard disk. The test table stores 10 million independently uniformly distributed 6-dimensional tuples on 211218 pages. Figure 7-8a shows a 35% measurement sequence, where A_2 to A_6 are the constant dimensions, whereas the restriction in A_1 is varied from 0% to 100%. As mentioned before, an FTS in DBMS1 is identical to an IOT on A_1 without any restriction in the index attribute. An IOT on A_2, A_3, A_4, A_5, or A_6 has to retrieve 35% of the database. This results in a response time 35% of that of an FTS. Actually the FTS is not constant, but slightly increases with a growing selectivity in A_1. The linearly growing result set causes linearly growing CPU time for result set processing and inter-process communication time for result set transfer. While this time is clearly visible for FTS and IOT on A_2, A_3, A_4, A_5, or A_6, it is also included in the response times of the other access methods. Intersection of secondary indexes in DBMS1 requires the retrieval of the 35% of the five secondary indexes on A_2 to A_6 and a certain percentage of the secondary index on A_1. After that an expensive intersection operation and a random access for each tuple of the result set are performed. Therefore SII is worse than tuple clustered access of the IOTs already for queries with a selectivity of less than 1%. The UB-Tree requires less time than any IOT, since it allows a tuple clustered access to the result set defined by the restrictions in all attributes, whereas an IOT on A_1 only utilizes the restriction on A_1 and therefore grows linearly on a much larger scale than the UB-Tree. The performance figures for the UB-Tree do hardly depend on the variable attribute: If A_1 was left constant and any other attribute is varied, the response time of the UB-Tree is similar to that reported in the figure (actually it depends on the number of recursive splits in that attribute; see [Fri97] for measurement charts or Chapters 3 and 6 for an analytical explanation).

Selectivity in A_1	**UB-Tree**	**IOT A_1**	**IOT A_2**	**SII**	**FTS**
20%	3,9 s	124,2 s	194,2 s	1890,3 s	458,9 s
40%	6,6 s	228,1 s	200,4 s	2120,9 s	477,3 s

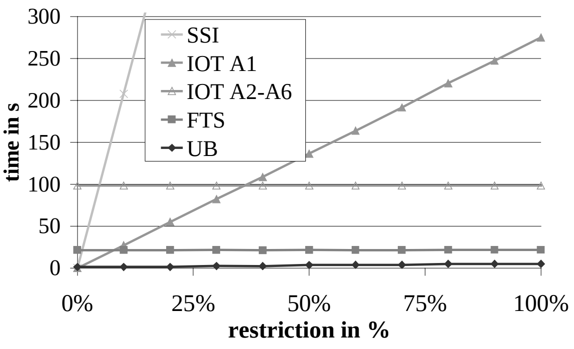
Table 7-4: Response times for 35%-measurements on DBMS1 with A_1 as variable attribute

Figure 7-8b shows a cube measurement sequence where the selectivity of each attribute is varied from 0% to 100% at the same time. For a selectivity of less than 8% in each dimension (an overall selectivity of $(8\%)^6 < 1/3.5 \cdot 10^{-6}$ for the query, i.e., a result set of about 3 tuples!) SII is preferable to an FTS, since the part of each B-Tree that needs to be retrieved as well as the result set are sufficiently small. Since the result set grows polynonially with the 6th power, the response time of the FTS grows with the 6th power due to CPU time for result set processing. This additional CPU time is also included in the response time of the other indexes. For a selectivity of 100% FTS, IOT, and UB-Tree take the same time to respond to the query, since the data is retrieved by tuple clustered access by all of these access methods. Up to a selectivity of 75% in each dimension (an overall selectivity of $(75\%)^6 = 17{,}7\%$, i.e., a result set of 1.7 million tuples) the UB-Tree has a significant performance advantage, since the restrictions in all attributes are used to limit the number of page accesses to answer the query.

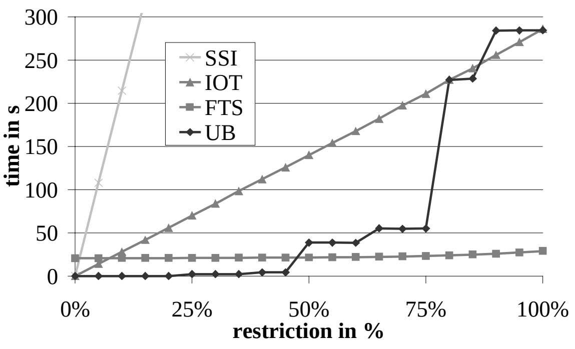
Selectivity in each dimension	**UB-Tree**	**IOT**	**SII**	**FTS**
20%	0,9 s	120,7 s	1235,1 s	449,8 s
40%	17,5 s	228,2 s	2753,3 s	475,9 s

Table 7-5: Response times for cube measurements on DBMS1

Summing up, our measurements indicate that the range query performance of UB-Trees on DBMS1 is more symmetrical than that of an IOT. It also shows a better absolute performance than SII, when a sufficient number of attributes is specified. In our 6-dimensional test database this is already true for 2 or 3 dimensions. The UB-Tree range query performance is on the average several orders of magnitude faster than IOTs and SII. We measured an increase in speed of several thousands compared to SSI and – depending on the restriction – between two and one-hundred compared to a compound index. Performing an index scan over the whole relation with a UB-Tree results in a performance similar to a scan over a clustered primary compound B-Tree (see also [Fri97]).



(a)



(b)

Figure 7-9: DBMS2

Figure 7-9 shows 35%-measurements and cube measurements on DBMS2 performed on a 4 CPU Pentium Pro 200 MHz with Windows NT 4.0. The test table stores 250 thousand independently uniformly distributed 6-dimensional tuples on 27194 pages. Besides the more efficient FTS and the use of a SSI instead of SII these measurement series are qualitatively identical to those of DBMS1 on Solaris 2.5.1 as described above. The only difference to DBMS1 is that DBMS2 does not perform an intersection of secondary indexes for this query. Using bitmap indexes resulted in a performance worse than any of the IOT. Instead, we measured the use of a single secondary index (SSI) on the most selective attribute. Further measurements on DBMS2 (varying tuple size, varying table size, comparison between NT and Solaris) showed the behavior as predicted by our cost model (see [Pie98]).

Selectivity in A_1	**UB-Tree**	**IOT A_1**	**IOT A_2**	**SSI**	**FTS**
20%	1,4 s	55,0 s	98,3 s	415,7 s	21,4 s
40%	2,4 s	108,6 s	98,5 s	725,1 s	21,3 s

Table 7-6: Response times for 35%-measurements on DBMS2 with A_1 as variable attribute

Selectivity in each dimension	**UB-Tree**	**IOT**	**SSI**	**FTS**
20%	0,1 s	55,8 s	428,4 s	20,9 s
40%	4,5 s	112,1 s	852,2 s	21,4 s

Table 7-7: Response times for cube measurements on DBMS2

7.4.3 Partial Range Queries

The following measurements illustrate the performance of the UB-Tree for partial range queries. The measurements were also performed with DBMS2 on a 4 CPU Pentium Pro 200 MHz with Windows NT 4.0. The test table stores 250 thousand independently uniformly distributed 6-dimensional tuples on 27194 pages.

While Section 7.4.2 restricted each of the *d* attributes of the UB-Tree to an interval smaller than the domain of an attribute, some dimensions are not restricted on this measurement. The measurement series here resembles a 35%-measurements series with the only difference that less than *d*-1 dimensions are restricted to 35%. The number *a* in the legend entry “UB *a* + 1” of Figure 7-10 shows how many dimensions are restricted to 35%. The other *d*-*a*-1 dimensions are not restricted. The graph shows a variation of *a* from 1 to 5 for the 6-dimensional UB-Tree (note that “UB 5 + 1” is the actual 35%-measurement series of the previous section). “IOT var” in the legend of the figure means that an IOT on the variable attribute of the 35%-measurement is used for answering the query, whereas “IOT 35%” resp. “IOT 100%” use an IOT on an attribute with a selectivity of 35% resp. 100% for processing the query. The measurement shows that as soon as two dimensions are restricted the UB-Tree is superior to IOTs, unless the restriction in the first attribute of the IOT is sufficiently small. The performance increase of the UB-Tree with a growing number of restricted dimensions shows that the UB-Tree indeed utilizes the restriction in all dimensions in order to reduce I/O. As soon as at least 4 out of 6 dimensions are restricted, the UB-Tree is also superior to an FTS even though the FTS of DBMS2 uses page clustering. The UB-Tree is also superior to an FTS, if at least 3 out of 6 attributes are restricted with a selectivity of less than 50% or 2 out of six attributes are restricted to a selectivity of less than 25%.

Figure 7-11 resembles a cube measurement series, where the “UB *a*” legend entry means that only *a* out of *d* attributes are restricted. “UB 6” of that figure is equivalent to the cube measurement series of the previous section .”IOT var” here means that an IOT on a variable dimension is used for query processing, whereas “IOT const” uses an IOT on a non-restricted attribute. Again the figure shows that further restricted attributes are used by the UB-Tree to reduce the response time for query processing, while FTS and IOTs cannot take any advantage of additional restrictions. An IOT on a variable attribute is superior to the UB-Tree, if only this attribute is restricted by the query (i.e., the query defines a hyperplane, only one dimension is restricted). Of course a specialized one-dimensional index like an IOT in this case is better than a multidimensional UB-Tree. However, as soon as at least two out of six attributes are restricted, the UB-Tree is superior to an IOT. As soon as at least 3 out of 6 attributes are restricted to a selectivity of less than 50%, the UB-Tree is also superior to an FTS even though the FTS of DBMS2 uses page clustering.

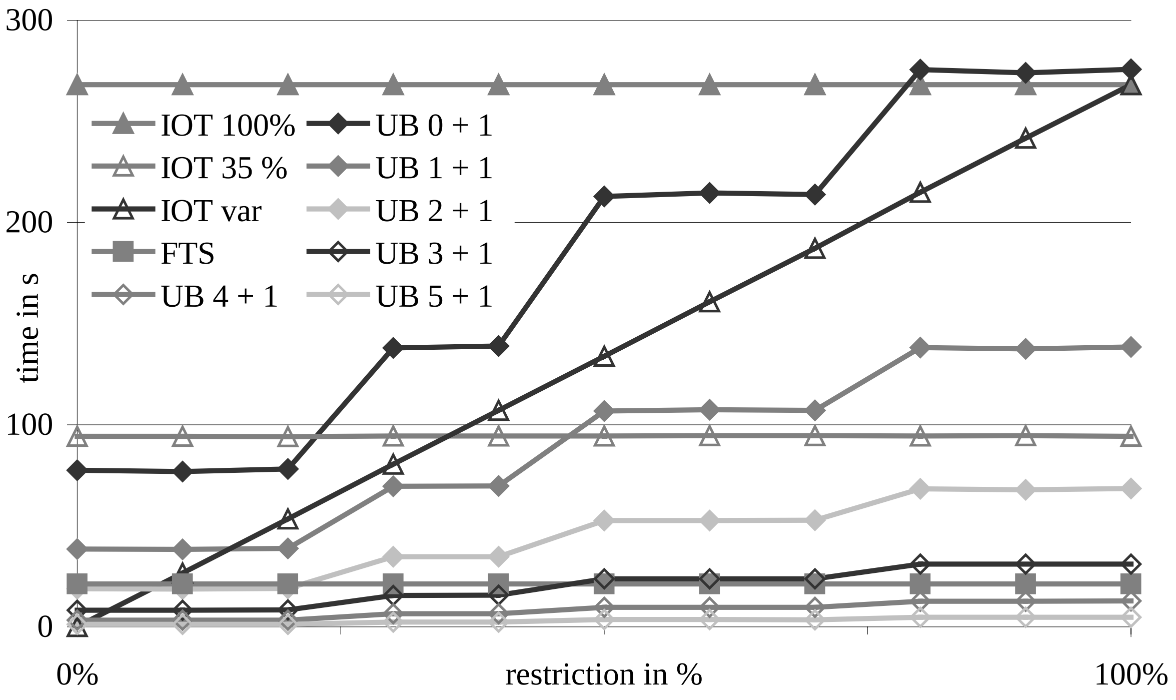


Figure 7-10: Varying the number of restricted dimensions for 35% restrictions (DBMS2)

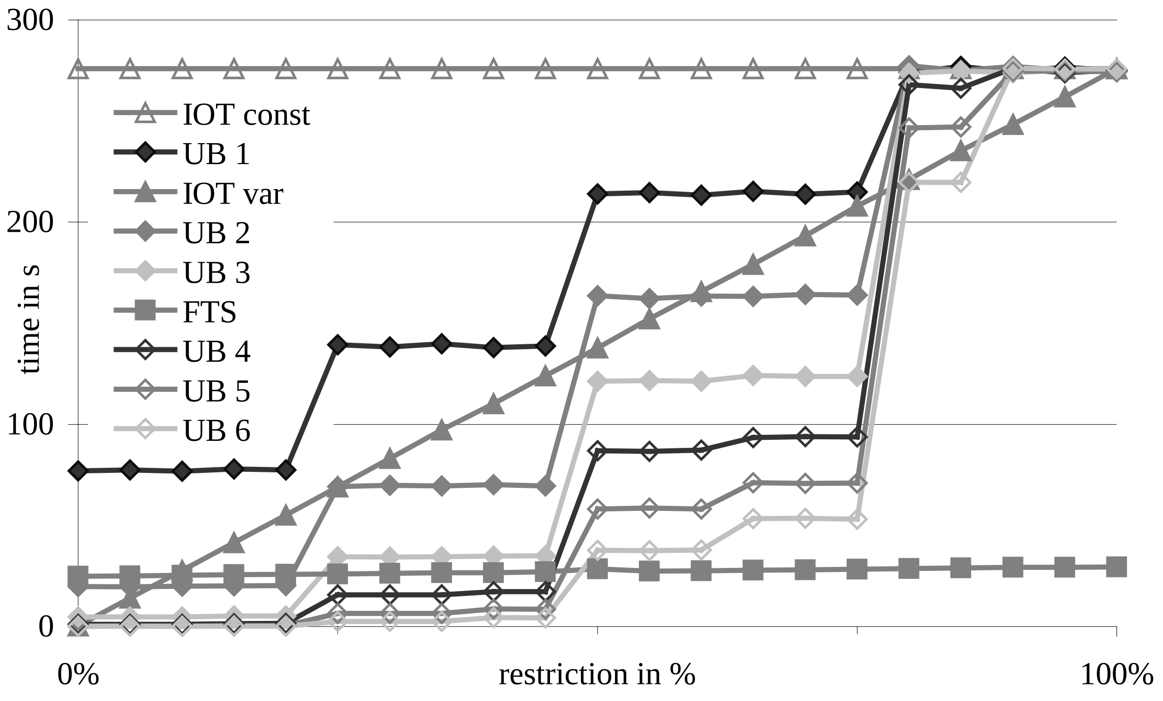


Figure 7-11: Varying the number of restricted dimensions for cube restrictions (DBMS2)

Change is not made without inconvenience, even from worse to better.

(Richard Hooker)

Chapter 8

Impacts on Relational Query Processing

UB-Trees and the Tetris algorithm can be used to accelerate almost any query processing operation: Relational queries or SQL queries consist of restrictions, projections, ordering, grouping and aggregation, and join operations. In the presence of multidimensional restrictions or sorting these operations are efficiently implemented by either using the range query algorithm or the Tetris algorithm. In this chapter we investigate the impacts of our approach on query processing in RDBMS. We present performance measurements for two application scenarios: We selected three queries of the TPC-D benchmark to show the potential of the UB-Tree range query algorithm and the Tetris algorithm. We then show the benefits of Multidimensional Hierarchical Clustering by performance measurements of queries in a star schema typical for data warehousing applications. The performance results reported in this chapter were measured for one of our project partners with our prototype implementation of UB-Trees on top of DBMS2. We compare the performance of UB-Trees to native query processing techniques of DBMS2, namely access via an index organized table (IOT), which essentially stores a relation in a clustered B*-Tree, and access via a full table scan (FTS) of an entire relation. In addition we measure the performance of a single secondary B*-Tree index (SSI) and of an intersection of multiple bitmap indexes (BII) to answer multidimensional range queries.

8.1 Relational Operations with UB-Trees and the Tetris-Algorithm

In accordance with the definition of O(n) (e.g., [AHU74]) we define the terms CPU-complexity, asymptotic CPU-complexity CPU(n), I/O complexity, asymptotic I/O-complexity IO(n), space complexity and asymptotic space complexity SPACE(n). These operations will be used to investigate the complexity of the operations of the relational algebra using UB-Trees with the range query algorithm and the Tetris algorithm.

Definition 8-1 (CPU-complexity, asymptotic CPU-complexity CPU(*n*)): The *CPU-complexity* of an algorithm is the CPU-time needed by an algorithm expressed as a function of the size of the input data. An algorithm with a CPU-complexity defined by a function $g(n)$ has an *asymptotic CPU-complexity* CPU($f(n)$), if there exists a constant c such that $g(n) \leq c \cdot f(n)$ for all but some finite (possibly empty) set of non-negative values for n.

Definition 8-2 (I/O-complexity, asymptotic I/O-complexity IO(*n*)): The *I/O-complexity* of an algorithm is the I/O-time (usually measured by the number of random disk accesses) needed by an algorithm expressed as a function of the size of the input data. An algorithm with a I/O-complexity defined by a function $g(n)$ has an *asymptotic I/O-complexity* IO($f(n)$), if there exists a constant c such that $g(n) \leq c \cdot f(n)$ for all but some finite (possibly empty) set of non-negative values for n.

Definition 8-3 (SPACE-complexity, asymptotic SPACE-complexity SPACE(*n*)): The *SPACE-complexity* of an algorithm is the temporary storage space needed by an algorithm expressed as a function of the size of the input data. An algorithm with a SPACE-complexity defined by a function $g(n)$ has an *asymptotic SPACE-complexity* SPACE($f(n)$), if there exists a constant c such that $g(n) \leq c \cdot f(n)$ for all but some finite (possibly empty) set of non-negative values for n.

8.1.1 Asymptotic complexity of the UB-Tree Range Query Algorithm

In the following we assume a multidimensional query box with the selectivities s_1, ..., s_d over a relation of T tuples stored on P pages. We consequently also assume Z-addresses to have a length of a bits.

Thus the range query algorithm has a CPU-complexity which depends on the number of Z-regions intersecting the query box. According to our cost functions of Section 6.1 this number is related to the selectivity of the restrictions of the query box. In addition the CPU-complexity of the range query algorithm linearly depends on the length of the Z-addresses. The Z-address length is identical to the length of the index attributes of a tuple (see Section 5.3). Thus the range query algorithm also depends linearly on the tuple size (see Section 5.7).

The I/O-complexity solely depends on the number of Z-regions intersecting the query box and only one Z-region needs to be stored to perform the range query algorithm. Thus the storage space complexity of the range query algorithm is constant with respect to the problem size.

Using the results of Section 6.1 and 5.7, the range query algorithm for a query box has an asymptotic CPU-complexity of

$$\mathrm{CPU}\left(a \cdot P \cdot \prod_{j=1}^{d} \hat{s}_j\right) \text{ bit operations,}$$

an asymptotic I/O-complexity of

$$\mathrm{IO}\left(P \cdot \prod_{j=1}^{d} \hat{s}_j\right) \text{ random disk accesses,}$$

and an asymptotic space complexity of

$$\mathrm{SPACE}(1).$$

8.1.2 Asymptotic complexity of the Tetris Algorithm

The Tetris algorithm has to retrieve the same number of Z-regions as the range query algorithm. The number of CPU-operations for determining the next Z-region in Tetris order are also identical to those of the range query algorithm. Thus from a complexity point of view the only difference between the Tetris algorithm and the range query algorithm is the storage complexity, which in this case is determined by the Tetris cache.

We again assume a multidimensional query box with the selectivities $s_1, ..., s_d$ over a relation of T tuples stored on P pages. We also assume a UB-Tree with a Z-address length of a bits. Then the Tetris algorithm to sort the query box with respect to attribute A_k has an asymptotic CPU-complexity of

$$\mathrm{CPU}\left(a \cdot P \cdot \prod_{j=1}^{d} \hat{s}_j\right),$$

an asymptotic I/O-complexity of

$$\mathrm{IO}\left(P \cdot \prod_{j=1}^{d} \hat{s}_j\right) \text{ random disk accesses,}$$

and an asymptotic space complexity of

$$\mathrm{SPACE}\left(\frac{P \cdot \prod_{j=1}^{d} \hat{s}_j}{l_k}\right) \text{ disk pages.}$$

In the above formula $l_k = n_k(d, P, y_k, z_k)$ is the number of slices of the query box in dimension k with respect to the multidimensional space partitioning (see Section 6.1).

8.1.3 Asymptotic complexity of Operations of Relational Query Processing

The UB-Tree range query algorithm may be used to efficiently process multi-attribute restrictions. The Tetris algorithm may be used to implement most operations of the relational algebra or SQL with linear I/O-complexity, if the Tetris cache size suffices to hold one processing slice of the Tetris algorithm [Bay97b]. This assumption is realistic, since our measurements showed that sorting 50 % of a 1.3 GB table only requires a cache size of 2.6 MB (cf. Section 8.2.1). The actual cache size depends on the multidimensional partitioning of the table and thus on the data distribution. While it is not possible to give an upper bound for the cache size (besides the relation size), experiments with various database sizes showed that usually a reasonable partitioning exists which keeps the cache size very low.

We denote the selection operations by σ, the join operation by $\bowtie$, grouping by γ, ordering by ω, set union by $\cup$, intersection by $\cap$ and difference by $\setminus$. These operations are implemented by operator trees processing several operators on tuple streams. Next to the access primitives *range* and *tetris* (which implement the UB-Tree range query algorithm and the Tetris algorithm) we use the operator *merge* to merge two sorted streams of tuples based on identity in an attribute set, the operator *remove-duplicate*, which eliminates duplicates of a sorted stream of tuples and *aggregate*, which performs grouping and performs the required aggregations. In addition we use the tuple stream operators *union*, *intersection* and *difference* that perform set union, intersection and difference on an ordered stream of tuples.

The range query algorithm and the Tetris algorithm can be used to implement the following operations of SQL or the relational algebra, since an efficient implementation of this algorithm may utilize a sorted stream of tuples:

- projecting *R* to *attr* with duplicate elimination: $\pi_{attr}(R)$
- sorting *R* with respect to *attr*: $\omega_{attr}(R)$
- grouping *R* with respect to *attr* and aggregating attributes with aggregation functions as specified in *agg*: $\gamma_{attr,agg}(R)$
- equi-joining *R* with *S* with respect to *attr*: $R \bowtie_{attr} S$
- set union of R and S: $R \cup S$
- set intersection of R and S: $R \cap S$
- set difference of R and S: $R \setminus S$

If multi-attribute restrictions on the relation(s) are used in combination with any of the above operations, these restrictions are utilized on the fly and thus reduce both the Tetris-cache size and number of page accesses necessary to perform the operation.

$\sigma_{cond}(R)$ = range$_{cond}(R)$ (selection)

$\pi_{attr}(\sigma_{cond}(R))$ = remove-duplicate(tetris$_{attr,cond}(R)$) (projection)

$\omega_{attr}(\sigma_{cond}(R))$ = tetris$_{cond}(R, attr)$ (ordering)

$\gamma_{attr,agg}(\sigma_{cond}(R))$ = aggregate(tetris$_{cond}(R, attr)$, agg) (grouping and aggregation)

$\sigma_{cond1}(R) \bowtie_{attr} \sigma_{cond2}(S)$ = merge(tetris$_{cond1}(R, attr)$, tetris$_{cond2}(S, attr)$) (join)

$\sigma_{cond1}(R) \cup \sigma_{cond2}(S)$ = union(range$_{cond1}(R)$, range$_{cond2}(S)$) (set union)

$\sigma_{cond1}(R) \cap \sigma_{cond2}(S)$ = intersect(range$_{cond1}(R)$, range$_{cond2}(S)$) (set intersection)

$\sigma_{cond1}(R) \setminus \sigma_{cond2}(S)$ = difference(range$_{cond1}(R)$, range$_{cond2}(S)$) (set difference)

Figure 8-1: Transformation rules for the implementation of algebraic operations

Figure 8-1 gives transformation rules for projection, ordering, grouping and aggregation and join in combination with multi-attribute restrictions, so that the Tetris operator can be used for efficient implementation of these operations. In the Figure *R*, *S* denote relations, *cond*, *cond1*, *cond2* denote conditions defining multidimensional intervals, *attr* denotes an attribute of *R* respectively *S* and *agg* denotes a specification of attributes and an aggregation function for the attributes. The rules given in this figure may be used as transformation rules for algebraic query optimization.

Applying the transformation rules of Figure 8-1, Table 8-1 lists the asymptotic CPU-, I/O- and space-complexity of the basic operations of relational query processing and opposes them to their complexity with a B-Tree implementation.

In the table we write $V = P \cdot \prod_{j=1}^{d} \hat{s}_j$ to denote the size of the result set of condition *cond* (analgously for V_R, V_S for $cond_R$ and $cond_S$). We assume the B-Tree to be built on the attribute *i* and denote the selectivity of that attribute by s_i. If the index attribute *i* is also the sort attribute *attr*, then $s_i = 0$ and $V = P$. We denote the length of a tuple in bits by *a*.

	UB-Tree/Tetris			B-Tree/Mergesort		
	CPU	I/O	SPACE	CPU	I/O	SPA-CE
$\sigma_{cond}(R)$	$O(V \cdot a)$	$O(V)$	O(1)	$O((P\cdot s_i+V\cdot\log V)\cdot a)$	$O(P\cdot s_i+V\cdot\log V)$	$O(V)$
$\pi_{attr}(\sigma_{cond}(R))$	$O(a\cdot V\cdot log\ V)$	$O(V)$	$O\left(\frac{V}{l_{attr}}\right)$	$O((P\cdot s_i+V\cdot\log V)\cdot a)$	$O(P\cdot s_i+V\cdot\log V)$	$O(V)$
$\omega_{attr}(\sigma_{cond}(R))$	$O(a\cdot V\cdot log\ V)$	$O(V)$	$O\left(\frac{V}{l_{attr}}\right)$	$O((P\cdot s_i+V\cdot\log V)\cdot a)$	$O(P\cdot s_i+V\cdot\log V)$	$O(V)$
$\gamma_{attr,agg}(\sigma_{cond}(R))$	$O(a\cdot V\cdot log\ V)$	$O(V)$	$O\left(\frac{V}{l_{attr}}\right)$	$O((P\cdot s_i+V\cdot\log V)\cdot a)$	$O(P\cdot s_i+V\cdot\log V)$	$O(V)$
$\sigma_{cond1}(R)$ $\bowtie_{attr}$ $\sigma_{cond2}(S)$	$O(a\cdot V_R\cdot log\ V_R)$ + $O(a\cdot V_S\cdot log\ V_S)$	$O(V_R+V_S)$	$O\left(\frac{V_R}{l_{attr}}+\frac{V_S}{k_{attr}}\right)$	$O((P_R\cdot s_i+V_R\cdot\log V_R)\cdot a)$ + $O((P_S\cdot s_i+V_S\cdot\log V_S)\cdot a)$	$O(P_R\cdot s_i+V_R\cdot\log V_R)$ + $O(P_S\cdot s_i+V_S\cdot\log V_S)$	$O(V)$
$\sigma_{cond1}(R)$ $\cup$ $\sigma_{cond2}(S)$	$O(a\cdot V_R)$ + $O(a\cdot V_S)$	$O(V_R+V_S)$	O(1)	$O((P_R\cdot s_i+V_R\cdot\log V_R)\cdot a)$ + $O((P_S\cdot s_i+V_S\cdot\log V_S)\cdot a)$	$O(P_R\cdot s_i+V_R\cdot\log V_R)$ + $O(P_S\cdot s_i+V_S\cdot\log V_S)$	$O(V)$
$\sigma_{cond1}(R)$ $\cap$ $\sigma_{cond2}(S)$	$O(a\cdot V_R)$ + $O(a\cdot V_S)$	$O(V_R+V_S)$	O(1)	$O((P_R\cdot s_i+V_R\cdot\log V_R)\cdot a)$ + $O((P_S\cdot s_i+V_S\cdot\log V_S)\cdot a)$	$O(P_R\cdot s_i+V_R\cdot\log V_R)$ + $O(P_S\cdot s_i+V_S\cdot\log V_S)$	$O(V)$
$\sigma_{cond1}(R)$ $\setminus$ $\sigma_{cond2}(S)$	$O(a\cdot V_R)$ + $O(a\cdot V_S)$	$O(V_R+V_S)$	O(1)	$O((P_R\cdot s_i+V_R\cdot\log V_R)\cdot a)$ + $O((P_S\cdot s_i+V_S\cdot\log V_S)\cdot a)$	$O(P_R\cdot s_i+V_R\cdot\log V_R)$ + $O(P_S\cdot s_i+V_S\cdot\log V_S)$	$O(V)$

Table 8-1: Complexities of relational operators

Thus UB-Trees and the Tetris algorithm have the potential to speed up any operation involving multi-attribute restrictions and sort operations. In Section 8.2 we will show performance measurements and comparisons for queries using some these operations. In Section 8.3 we will investigate a special variant of a *multiway join-operation*, the so-called *star-join*, and reduce it to multi-attribute restrictions with the technique of multidimensional hierarchical clustering as described in Section 5.3.4.

8.2 Complex Queries on Generated Data: The TPC-D Benchmark

We analyzed the entire TPC-D benchmark [TPC97] for the usability of multidimensional access methods. From our analysis we expect that 12 out of 17 queries will benefit from multidimensional indexing techniques and the Tetris algorithm. The five queries that will not benefit from multidimensional indexes either only restrict a single attribute without complex joins/sort operations or retrieve a result set whose size makes an FTS preferable to any access method. To show the performance gain of UB-Trees and the Tetris algorithm we selected the TPC-D queries Q3, Q4 and Q6.

We used a SUN ULTRA SPARC II with 512 MB main memory and an array of five 4 GB hard disks with an average positioning time of 8ms and a transfer rate of 0.7ms per page to generate the ORDER, LINEITEM, and CUSTOMER tables (cf. Figure 2-2) for several scaling factors of the TPC-D benchmark. Actually the Tetris performance is even better than reported in this section: The measurements were conducted with UB-Trees emulated on top of DBMS2 and are compared against IOTs and FTS integrated into the DBMS2 kernel.

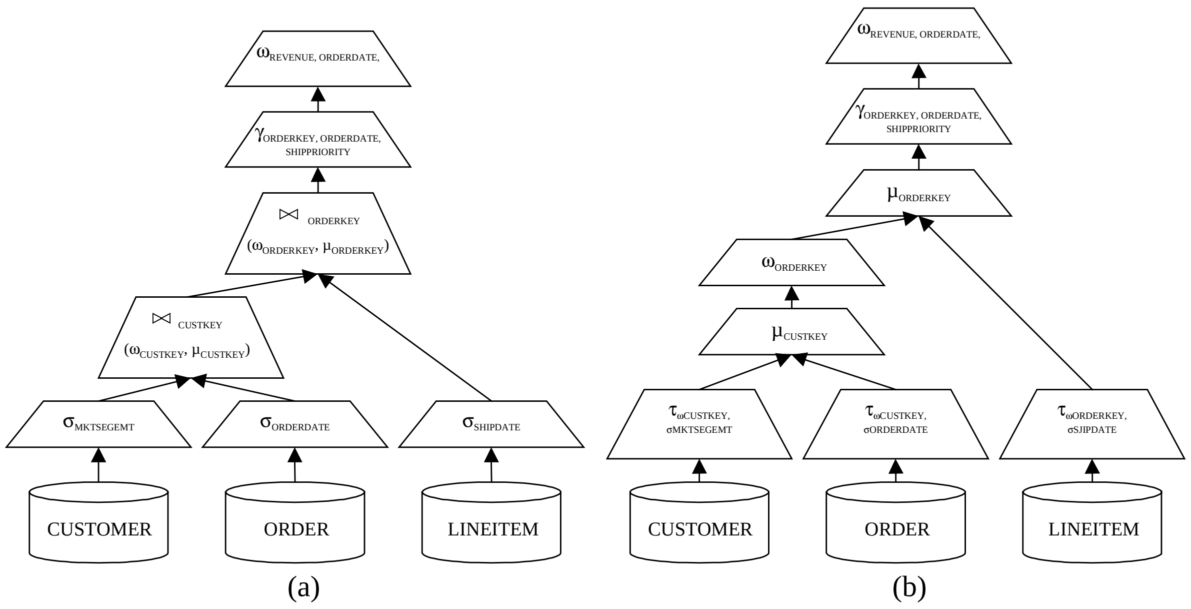
8.2.1 Joins and Restrictions

Query Q3 (cf. Figure 8-2) of the TPC-D benchmark is a shipping priority query, which retrieves the shipping priority and potential revenue of the orders having the largest revenue among those that had not been shipped as of a given data. This query consists of restrictions and join operations involving three relations and is efficiently processed by the Tetris algorithm.

```
SELECT L_ORDERKEY, SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS REVENUE,
     O_ORDERDATE, O_SHIPPRIORITY
FROM CUSTOMER, ORDER, LINEITEM
WHERE
     C_MKTSEGMENT = ’FOOD’ AND
     C_CUSTKEY = O_CUSTKEY AND
     L_ORDERKEY = O_ORDERKEY AND
     O_ORDERDATE < DATE 1.5.98 AND
     L_SHIPDATE > DATE 1.6.98
GROUP BY L_ORDERKEY, O_ORDERDATE, O_SHIPPRIORITY
ORDER BY REVENUE DESC, O_ORDERDATE
```

Figure 8-2: Query Q3 of the TPC-D benchmark

The operator tree for Q3 which is generated by a standard RDBMS like DBMS2 is illustrated in Figure 8-3a. The query is processed by first applying the restrictions on each table and then performing a hash join or a sort-merge join on the intermediate result. The join order of Figure 8-3a is due to the fact that the LINEITEM relation is four times larger than the ORDER relation and 40 times larger than the CUSTOMER relation. The intermediate result of the second join is used for grouping with aggregation and final ordering.



$\omega_{REVENUE, ORDERDATE,}$
$\gamma_{ORDERKEY, ORDERDATE, SHIPPRIORITY}$
$\bowtie_{ORDERKEY}$ $(\omega_{ORDERKEY}, \mu_{ORDERKEY})$
$\bowtie_{CUSTKEY}$ $(\omega_{CUSTKEY}, \mu_{CUSTKEY})$
$\sigma_{MKTSEGEMT}$
$\sigma_{ORDERDATE}$
$\sigma_{SHIPDATE}$
CUSTOMER
ORDER
LINEITEM
(a)
$\mu_{ORDERKEY}$
$\omega_{ORDERKEY}$
$\mu_{CUSTKEY}$
$\tau_{\omega CUSTKEY, \sigma MKTSEGEMT}$
$\tau_{\omega CUSTKEY, \sigma ORDERDATE}$
$\tau_{\omega ORDERKEY, \sigma SJIPDATE}$
(b)

Figure 8-3: Operator trees for Q3

With UB-Trees on CUSTOMER(CUSTKEY, MKTSEGMENT), ORDER(ORDERKEY, CUSTKEY, ORDERDATE), and LINEITEM(SHIPDATE, ODERKEY) Figure 8-3b and Figure 8-4 illustrate the Tetris operator $\tau_{\sigma,\omega}$, which combines selection and sorting. Reading the restricted part of each relation in sort order of the join attribute causes a sorted stream of tuples. This stream is transferred to the merge operator μ and processed further.

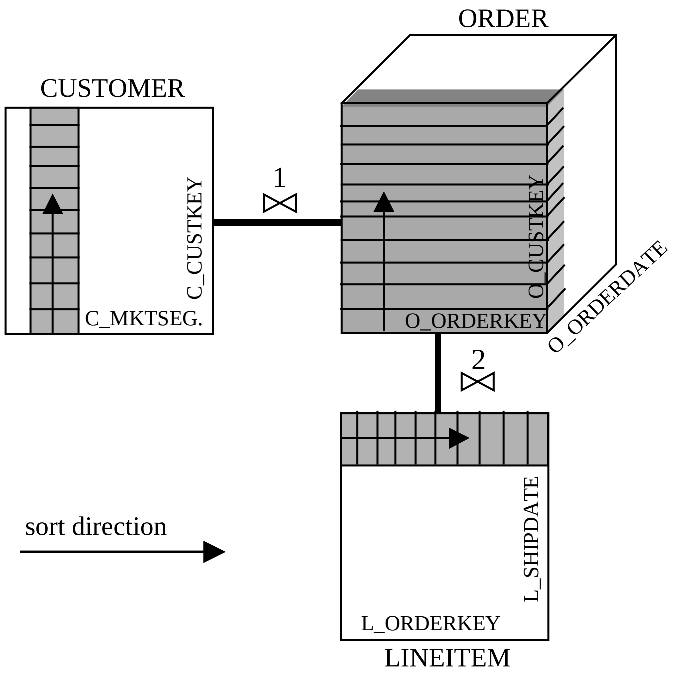


Figure 8-4: Processing Q3 with the Tetris algorithm

We measured the sorted table accesses of query Q3 for different TPC-D scaling factors (SF) from 0.1 to 1 (SF = 1 means a size of 1GB for LINEITEM). We do not want to enter the debate whether sort-merge joins or hash joins perform better [Mer81, DKO+84]. We chose a large main memory for our test environment, since according to [CHH+91] sort-merge join and hash join have a similar performance for computer systems with large main memories. Consequently we use a sort-merge join because this method is easier to handle by our test environment.

Since the LINEITEM table is the major bottleneck for Q3, we focus on this relation for our performance comparison. We created four instances of LINEITEM: an IOT on SHIPDATE, an IOT on ORDERKEY and a relation with secondary indexes on each restricted or sorted attribute. The optimizer favored an FTS over secondary indexes, which our theoretical considerations and measurements proved to be the right decision (forcing DBMS2 to process Q3 with a secondary index on SHIPDATE or ORDERKEY took more than 6 hours for SF = 1). We therefore exclude secondary indexes from further considerations.

Figure 8-5 and Table 8-2 show that the Tetris algorithm for UB-Trees is most preferable to answer this query. The 50% restriction on SHIPDATE is not selective enough for an IOT on SHIPDATE to be competitive. The presorted IOT on ORDERKEY does not require a merge sort and therefore shows response times similar to an FTS with merge sort. Using Tetris for sorting LINEITEM is more than three times faster than FTS or any IOT. The first response of Tetris is already produced after few seconds, two to three orders of magnitude faster than with FTS or any IOT. While the intermediate storage requirements of Tetris are not exactly zero as for an IOT on ORDERKEY, they are extremely low: Compared to an FTS or an IOT on SHIPDATE they are several orders of magnitude lower.

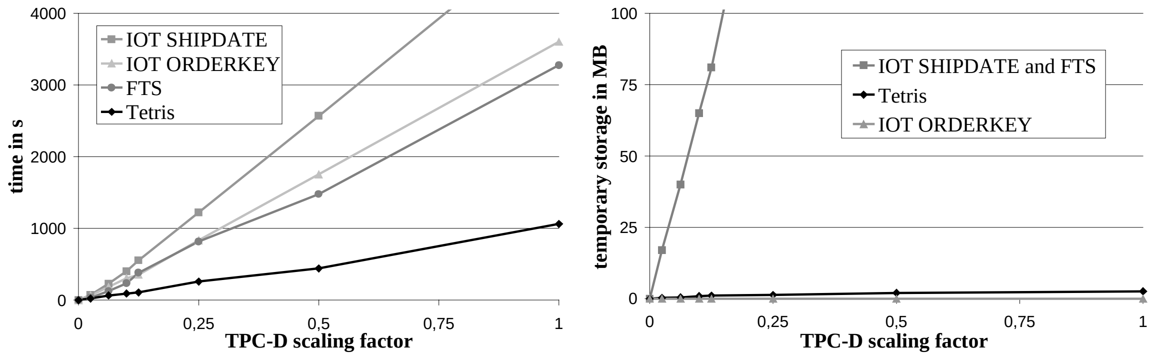


Figure 8-5: Response times and temporary storage for sorting 50 % of LINEITEM for Q3

Since for FTS and IOT on SHIPDATE storage requirements grow linearly with *tablesize*, the main memory is exceeded soon. To conduct our measurements we had to enlarge the temporary DBMS2 tablespaces several times. In contrast to that the Tetris cache grows with $\sqrt{tablesize}$ (cf. Section 6.4.1) and fits into the main memory of current computer systems even for table sizes of several Terabytes.

Table Size Scaling Factor (SF)	33 MB (0.025)	81 MB (0.0625)	131 MB (0.1)	163 MB (0.125)	326 MB (0.25)	651MB (0.5)	1302MB (1)
Tetris 1st response	0.3s	0.5s	0.7s	1,1s	1,3s	1,3s	3,3s
Tetris Slices	64	128	128	128	256	256	512
Time IOT ORDERKEY	64.7s	184.3s	306.7s	356.2s	834.3s	1753.6s	3604.1s
Time IOT SHIPDATE	72.5s	226.9s	401.3s	554..3s	1223.7s	2569.8s	5286.4s
Time FTS-Sort	34.1s	126.7s	234.0s	381.1s	816.5s	1479.4s	3276.4s
Time Tetris	23.1s	64.4s	92.5s	106.2s	257.5s	441.2s	1062.2s
Cache Tetris	0.3.MB	0.3MB	0.9MB	1.1MB	1.4MB	2.1MB	2.6MB
Temp Storage IOT/FTS	17MB	40MB	65MB	81MB	183MB	326MB	751MB

Table 8-2: Interactive response times and cache sizes for sorting 50 % of LINEITEM

8.2.2 Joins, Grouping, and Restrictions

The query Q4 (cf. Figure 8-6) of the TPC-D benchmark is an order priority checking query, i.e., it counts the number of orders placed in a given quarter of a given year in which at least one line item was received by the customer later than its committed date. The query lists the count of such orders for each order priority sorted in ascending priority order. Since Q4 involves restrictions, joins, and grouping, it is efficiently supported by the Tetris algorithm.

```
SELECT O_ORDERPRIORITY, COUNT(*) AS ORDER_COUNT
FROM ORDER
WHERE
 O_ORDERDATE >= DATE '[date]' AND
 O_ORDERDATE < DATE '[date]' + INTERVAL '3' MONTH AND
 EXISTS (   SELECT *
       FROM LINEITEM
       WHERE
             L_ORDERKEY = O_ORDERKEY AND
             L_COMMITDATE < L_RECEIPTDATE )
GROUP BY O_ORDERPRIORITY
ORDER BY O_ORDERPRIORITY
```

Figure 8-6: Q4 of the TPC-D benchmark

Assuming a three dimensional organization of ORDER (ORDERDATE, ORDERPRIORITY, ORDERKEY) and LINEITEM (COMMITDATE, RECEIPTDATE, ORDERKEY), query processing with the Tetris algorithm is shown in Figure 8-7. Q4 groups the restricted ORDER table depending on tuple existence in the LINEITEM table. Efficiently processing this query means processing ORDER in ORDERKEY order while using the 3.5%-restriction on ORDERDATE. To evaluate the existential restriction, LINEITEM is processed in ORDERKEY order and semi-joined to ORDER. The Tetris-algorithm can be used to process the triangular search space defined by COMMITDATE < RECEIPTDATE in ORDERKEY order. Processing each ORDERDATE-slice in ORDERPRIORITY order reduces the number of CPU operations, since groups can be built without comparisons. When processing the last ORDERDATE slice, completing an ORDERPRIORITY slice allows one to transfer the corresponding group immediately to the user.

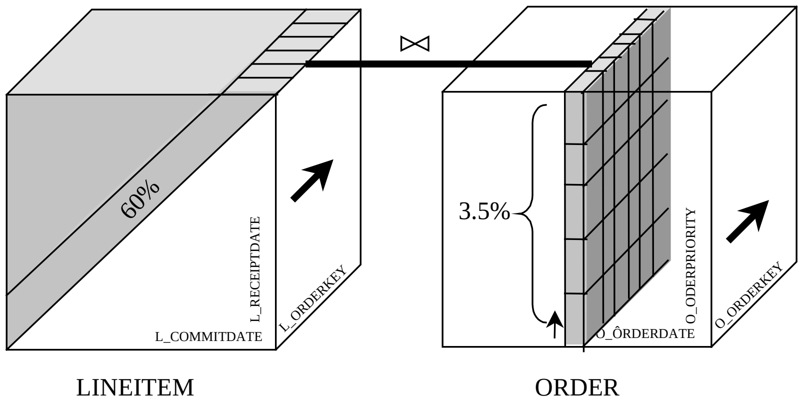


Figure 8-7: Processing Q4

We just report the response times and cache sizes of sorting ORDER in Figure 8-8 and Table 8-3, since the enhancement of the Tetris algorithm for non-rectangular query spaces has not been implemented yet. The restrictions on ORDER are selective enough for an IOT on ORDERDATE to be superior to FTS and IOT on ORDERKEY. In accordance with our cost functions the sudden increase of the FTS for SF = 0.5 is due to the fact that at this point the main memory is not sufficient anymore to sort the result internally. The Tetris algorithm is superior to FTS and any IOT, since it utilizes restrictions and sorts the data at the same time. Even for this quite selective ORDERDATE restriction the Tetris algorithm is more than three times faster than the IOT on ORDERDATE. Tetris also is 11 times faster than an FTS and about 30 times faster than an IOT on ORDERKEY.

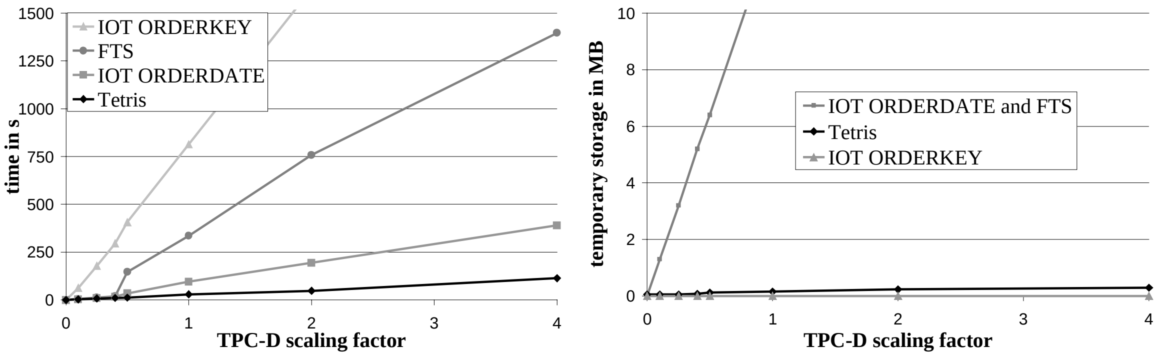


Figure 8-8: Responses time and temporary storage for sorting 3.5% of ORDER for Q4

As predicted by our cost functions (cf. Section 6.4.1), the Tetris cache of Figure 8-8 is more than 60 times lower than the intermediate storage of an IOT on ORDERDATE or FTS. Even for a ORDER table of 1.5GB (SF = 4) the 0.3 MB Tetris cache easily fits into main memory.

Table Size **Scaling Factor (SF)**	**32 MB** **(0.1)**	**79 MB** **(0.25)**	**131 MB** **(0.4)**	**161 MB** **(0.5)**	**322 MB** **(1)**	**750MB** **(2)**	**1498MB** **(4)**
Tetris 1st response	0,2s	0,4s	0,1s	0,1s	0.1s	0.2s	0.3s
Tetris Slices	64	128	128	128	256	256	512
Time IOT ORDERKEY	62.0s	178.4s	295.9s	406.9s	813.8s	1627.5s	3254.9s
Time IOT ORDERDATE	3.2s	9.2	16.8s	34.3s	95.4s	194.2s	390.4s
Time FTS-Sort	5.2s	12.5s	19.9s	146.4s	335.2s	758.6s	1396.7s
Time Tetris	3.3s	7.8	10.5s	12.2s	29.7s	47.8s	113.9s
Cache Tetris	0.1MB	0.1MB	0.2MB	0.2MB	0.2MB	0.2MB	0.3MB
Temp Storage IOT/FTS	1.3MB	3.2MB	5.2MB	6.4MB	12.9MB	30.1MB	60.1MB

Table 8-3: Interactive response times and cache sizes for sorting 3.5% of ORDER

8.2.3 Multi-attribute Restrictions

Query Q6 (cf. Figure 8-9) of the TPC-D benchmark is a forecasting revenue query, which lists the amount by which the total revenue would have increased if the discounts had been eliminated for line items with a quantity less than a given quantity in a given year with a discount deviating 0.01 from a given discount. The UB-Tree range query algorithms may be used to efficiently process this query involving multi-attribute restrictions and aggregations.

```
SELECT SUM(L_EXTENDEDPRICE*L_DISCOUNT) AS REVENUE
FROM LINEITEM
WHERE
    L_SHIPDATE >= [date] AND
    L_SHIPDATE <= [date] + INVERVAL 1 YEAR AND
    L_DISCOUNT BETWEEN [discount] –0.01
         AND [discount] + 0.01 AND
    L_QUANTITY < [quantity]
```

Figure 8-9: Query Q6 of the TPC-D benchmark

Q6 is processed by either using an IOT on SHIPDATE to materialize the result and then check the conditions on DISCOUNT and QUANTITY, or perform an FTS, if no such index exists. Performing an index intersection on three secondary B^*-Trees is not very efficient, since the selectivity of an individual attribute is relatively low (20% for SHIPDATE, 33% for DISCOUNT and 50% for QUANTITY). An intersection of bitmap indexes is not a good choice either, since the number of distinct values for SHIPDATE, DISCOUNT, and QUANTITY is quite high. Since 1/30th of all tuples of LINEITEM satisfy the restrictions of Q6, 200k tuples have to be retrieved to process the query for SF = 1. Bitmap indexes and secondary B-Trees do not cluster the data. Therefore an FTS is preferable to both access methods. Multidimensional indexes cluster data symmetrically with respect to all index attributes. With 8kB pages 80 tuples of the LINEITEM relation are stored together on one page. Accessing 200k tuples then means 2.5k random disk accesses. Thus it makes sense to use a multidimensional index for this type of query.

For Q6 we created five instances of LINEITEM, namely a UB-Tree, an IOT on each restricted attribute, and a table with three secondary B-Trees, one on each restricted attribute. As expected it was not possible to make the optimizer perform an index intersection. The optimizer always preferred an FTS instead. Forcing the optimizer to use a single secondary index on SHIPDATE (the most selective attribute) was much less efficient than an FTS. Since this verifies our theoretical expectations, as before we exclude secondary indexes also from our performance comparison for Q6.

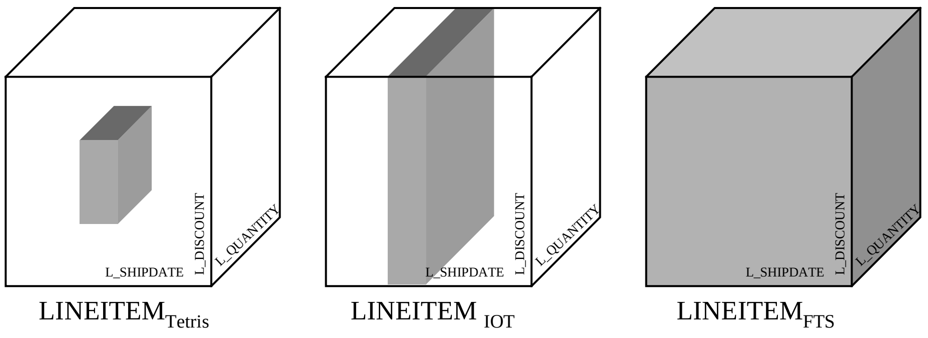


Figure 8-10: Processing Q6 with UB-Tree, IOT and FTS

The shaded part of Figure 8-10 shows the part of LINEITEM that is retrieved by the Tetris algorithm, the IOT on SHIPDATE, and the FTS in order to process Q6. Although an FTS retrieves the entire relation, prefetching strategies substantially reduce the number of random accesses and make the FTS superior to any IOT.

Table Size **Scaling Factor (SF)**	**33 MB** **(0.025)**	**81 MB** **(0.0625)**	**131 MB** **(0.1)**	**163 MB** **(0.125)**	**326 MB** **(0.25)**	**651MB** **(0.5)**	**1302MB** **(1)**
Time IOT QUANTITY	43,6s	109,2s	180,1s	225,2s	460,7s	921,4s	1842,8s
Time IOT DISCOUNT	31,2s	78,3s	126,4s	158,2s	339,2s	678,4s	1356,8s
Time IOT SHIPDATE	21,2s	53,7s	81,6s	102,1s	208,1s	416,3s	832,5s
Time FTS	5,2s	12,1s	19,2s	23,8s	47,7s	93,9s	187,6s
Time UB-Tree	1,1s	2,5s	4,5s	5,8s	12,0s	21,3s	30,5s

Table 8-4: Interactive response times for Q6

Table 8-4 and Figure 8-11 again show the superiority of a multidimensional organization over classical access methods by a sixfold speedup of the Tetris algorithm over an FTS and by a speedup of two to three orders of magnitude over any IOT. The results also show that an FTS is superior to any one dimensional index, since the FTS uses page clustering whereas indexes only use tuple clustering. In accordance with our cost formulas from Section 6.3, the restriction in any dimension is not selective enough for an IOT to make up the page clustering advantage of the FTS. However, a multidimensional organization of the table with a UB-Tree utilizes the restriction in all dimensions and thus clearly outperforms the FTS. In addition, an FTS puts an enormous load on the system in both I/O-resources and CPU-resources, since each tuple of the relation is retrieved and processed in main memory (cf. [Pie98]). Thus, especially for multi-user environments indexes may be preferable to an FTS because of concurrency considerations.

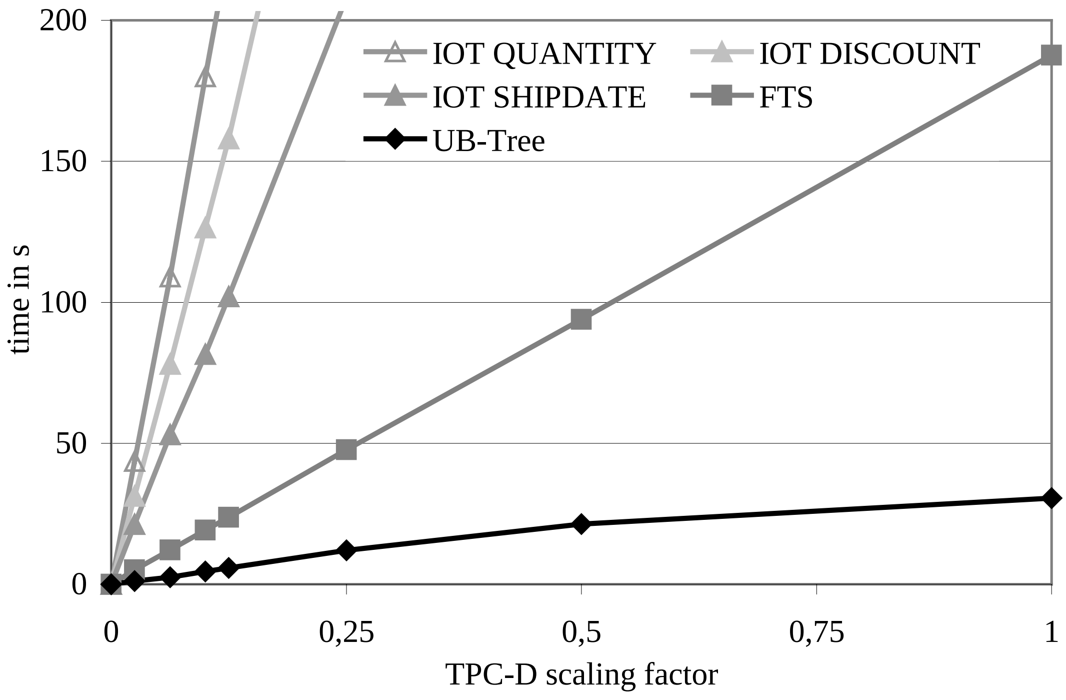


Figure 8-11: Performance of Q6

8.2.4 Summary: TPC-D Performance Measurements

Our performance measurements of three TPC-D queries have shown that UB-Trees and the Tetris algorithm are superior to one-dimensional access methods with respect to both response time and systems resources for storing intermediate results. With the Tetris algorithm a new operator may be introduced into query processing, the so-called Tetris operator. This operator combines the evaluation of multi-attribute restriction with a sort operation in one processing step, if a relation is organized by a multidimensional index.

Usually no more than 3 to 6 foreign keys are used to describe the foreign keys relationships between tables. For these dimensionalities only I/O-time linear in the size of the result set and sublinear temporary storage are necessary to perform the Tetris algorithm. In contrast to a merge-sort algorithm results are produced in a continuous flow of operation. Therefore sorting is no longer a blocking operation. Compared to existing techniques, the first results are available much earlier and thus allow better interactive response times and better internal pipelining of the data. The benchmark results for three queries of the TPC-D benchmark show speedups of up to two orders of magnitude in response time. Depending on the query, temporary storage requirements are reduced by several orders of magnitude

.

8.3 A Real World Data Warehouse: The Juice & More Benchmark

The most established relational data models for data warehousing applications are the *star schema* and the *snowflake schema*. In both approaches there is a central fact table that contains the measures and the dimension tables are situated around it. The connection between a fact tuple and the corresponding dimension members is realized via foreign key relationships. In the star schema the dimension tables are completely denormalized while in the snowflake schema they may be normalized. Queries usually contain restrictions on multiple dimension tables (e.g., only sales for specific customer group and for a specific time period are asked) that are then used as restrictions on the usually very large fact table. This operation (*star join*) is typical for such models. In ROLAP hierarchies are usually modeled implicitly by a set of attributes $A_1, ..., A_n$ where A_i corresponds to hierarchy level i.

In this section we investigate, how our technique of multidimensional hierarchical clustering of Section 5.3.4 may be used to accelerate star-joins, the most frequent operation of query processing for relational data warehouses.

We use the schema of the beverages supplier 'Juice & More', a real customer of one of our project partners[18]. In the data warehouse of 'Juice & More' data is organized along the following four dimensions: CUSTOMER, PRODUCT, DISTRIBUTION and TIME. Figure 8-12a shows the hierarchies over the dimensions (the number in parentheses specifies the maximal number of level members).

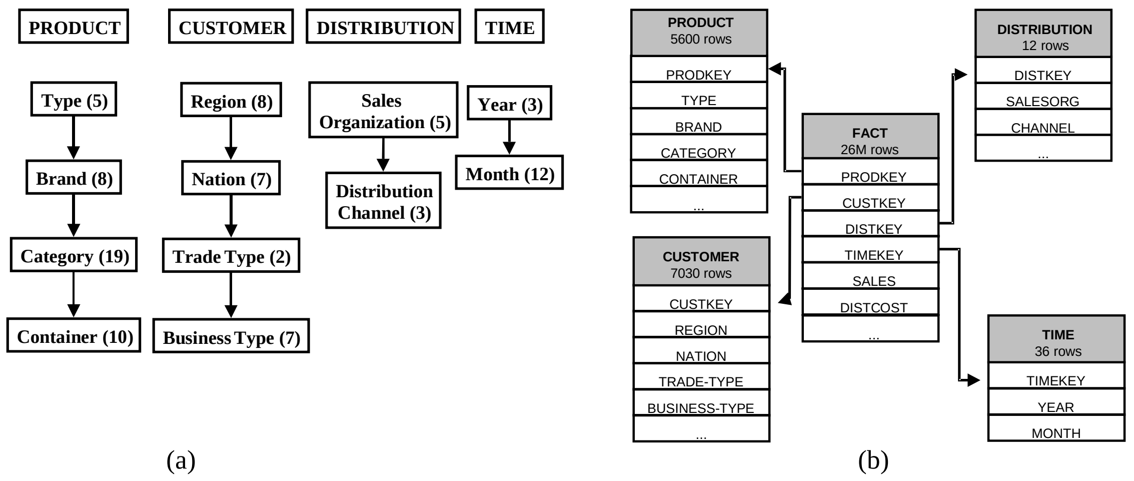


Figure 8-12 Hierarchies in the 'Juice & More' schema and the corresponding star schema

The ROLAP data model for the 'Juice & More' schema (Figure 8-12b) is a typical star schema with one fact table FACT and a table for each of the 4 dimensions. Let 'SALES' and 'DISTCOST' be some of the measures in the fact table. We used the methodologies of

[18] The company and the data presented here has been made anonymous.

surrogates and multidimensional hierarchical clustering as described in Section 5.3.4 for clustering the fact table of ‘Juice & More’ with UB-Trees.

In the following we describe some example queries involving star joins for the ‘Juice & More’ schema. These queries were taken from the real-world decision support system of ‘Juice & More’. The database schema and data of ‘Juice & More’ are real world data which we obtained from our project partners. Thus next to the investigation of multidimensional hierarchical clustering this section is interesting from a second point of view: The highly skewed data distribution of the ‘Juice & More’ will prove that UB-Trees and the Tetris algorithm are not only applicable to laboratory environment tests with generated data, but also prove their efficiency in the practical application scenarios. In order to show the highly skewed data distribution we included an entire section displaying the one-dimensional data distribution of ‘Juice & More’ for every dimension.

We then present measurements performed with our prototype implementation of the UB-Tree on top of DBMS2. For the evaluation of our clustering technique we defined a benchmark with 36 queries. In comparison we also conducted measurements with native DBMS2 access methods: full table scan (FTS) and bitmap indexes (BII). For these measurements we used a completely denormalized fact table, that is, no additional joins had to be performed to answer the queries. The bitmap indexes were created on each hierarchy level. We did not include secondary indexes in our comparison measurements because earlier experiments showed that they are neither competitive to the UB-Tree nor to FTS or BII [MZB99].

8.3.1 Queries on the ‘Juice & More’ Schema

In the following we present typical queries that are taken from real applications for the schema given in the previous section. We will use these queries to illustrate our approach and we will present performance measurements for exactly these queries in Section 8.3.4.

Query 1 (Q1, cf. Figure 8-13) computes the sales for a given product group (TYPE and BRAND specified as (X1, X2)) and a given customer group (NATION and REGION specified as (Y1, Y2)) for the months from October to December of 1993.

```
SELECT   SUM(Sales)
FROM     Fact F, Customer C, Product P, Time T
WHERE    F.ProdKey = P.ProdKey AND F.CustKey = C.CustKey AND
         P.Type = X1 AND P.Brand = X2 AND
         C.Region = Y1 AND C.Nation = Y2 AND
         F.TimeKey = T.TimeKey AND T.Year = 1993 AND
         T.Month >= October AND T.Month <= December
```

Figure 8-13: Time Interval (Q1)

Query 2 (Q2, cf. Figure 8-14) calculates the cost of distribution of the products of type X for each distribution channel.

```
SELECT   SALESORG, CHANNEL, SUM(DistCost)
FROM     Fact F, Distribution D, Product P
WHERE    F.DistKey = D.DistKey AND
         F.ProductKey = P.ProductKey AND
         P.Type = X
GROUP BY D.SalesOrg,D.Channel
```

Figure 8-14: Distribution cost (Q2)

Query 3 (Q3, cf. Figure 8-15) restricts all dimensions on the first level of the hierarchies.

```
SELECT   SUM(SALES)
FROM     Fact F, Distribution D, Product P, Customer C, Time T
WHERE    F.DistKey = D.DistKey AND F.TimeKey = T. TimeKey
             AND F.CustKey = C.CustKey AND F.ProdKey = P.ProdKey
             AND P.Type = t AND D.SalesOrg = s AND T.Year = y
             AND C.Region = r
```

Figure 8-15: Partial match query in the first hierarchy level (Q3)

8.3.2 Data Distribution

The data of 'Juice & More' is real world data from one of our project partners. In contrast to the data distributions used for most of the performance analyses and measurements in the previous sections, the 'Juice & More' data distribution of both the fact table and the dimension tables is highly skewed: The dimensions neither distributed uniformly nor are independent. The original fact table consisted of 823.464 tuples (about 175 MB). To get a realistic large data cube, the fact table was enlarged to 26.350.848 tuples (about 5,6 GB). Our project partner implemented an augmentation algorithm with minimal impact on the data distribution (see [Pie98]).

In the following we show some charts which describe the one-dimensional data distribution for each dimension. However, we once more stress that the dimensions are not independent (e.g., some customers always order the same subset of products, some customers or products only exist for a certain time, etc.). Thus in general, the overall selectivity of a query restricting several dimensions is *not* the product of the selectivities of the one-dimensional restrictions (which is shown in the following charts). We will see this deviation in Section 8.3.3.

The fact table of 'Juice & More' stores several measures (e.g., distribution cost, sales) aggregated on a daily basis with respect to the dimensions time, customer, product and distribution. Since the data is sensitive real-world business data, it is not possible to show the labels/names of the hierarchy members in the charts.

8.3.2.1 The Time Dimension

The time dimension of 'Juice & More' consists of a two-level hierarchy of months and years. The test data stored the years from 1993 to 1995. As shown in Figure 8-12a, the days of the time dimension are organized by a two level hierarchy (year and month). Figure 8-16 shows the cumulated data distribution of the fact table with respect to the time dimension grouped by year and month. The horizontal axis displays the hierarchy members, with "All" at the very bottom (i.e., the lowest level), the years 1993, 1994, and 1995 in the middle and the twelve months for each year above the year. The arrows in the horizontal axis indicate the relationship between the members of neighboring hierarchy levels.

Thus the fact table of 'Juice & More' is almost uniformly distributed with respect to the time dimension. The minimum number of facts for one month is 2,70% of the fact table (in January 1993, December 1993 and January 1995), whereas the maximum number of facts per month is 2,83% (in March of each of the three years). Thus with a multidimensional clustering using 5 split levels (cf. sections 3.10 and 6.1) restrictions to one month in the time dimension (=1/36) can be expected to reduce the amount of data to around $1/2^5 = 1/32$.

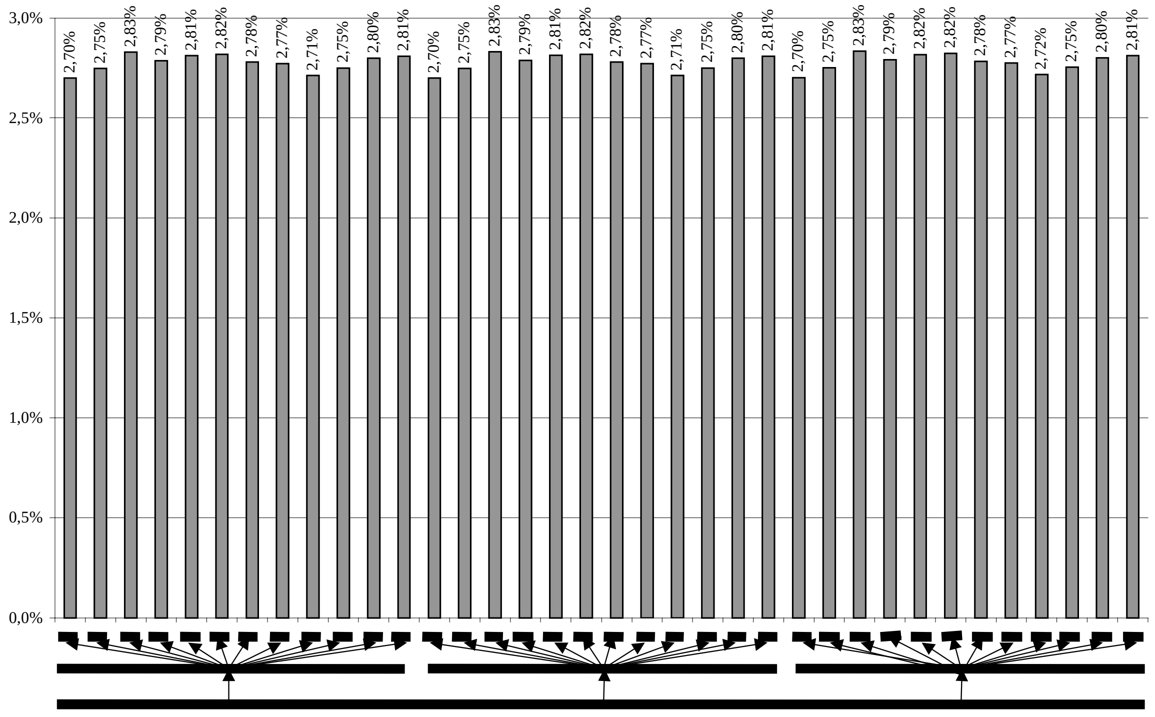
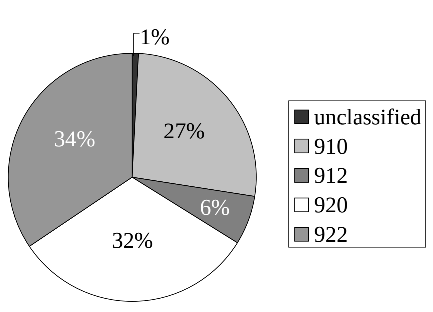


Figure 8-16: Data distribution of the time dimension

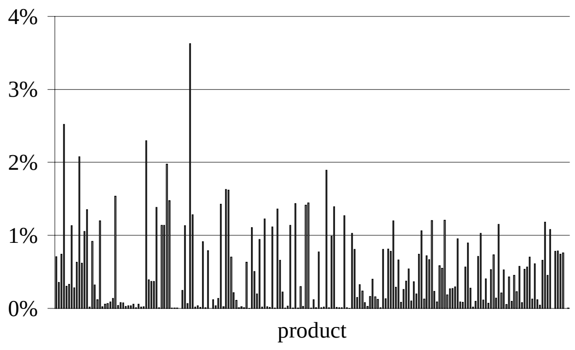
8.3.2.2 The Product Dimension

The top level of the hierarchy on product has five entries. The data distribution is quite skewed, there are three product groups to which 93% of all tuples of the fact table belong. 1% of the data is unclassified. The distribution of the first level of the product hierarchy is illustrated in Figure 8-17a. Multidimensional hierarchical clustering as described in 5.3.4

ensures that a restriction in the first hierarchy level will result in a 1%, 27%, 6%, 32% respectively 34% reduction of I/Os which are necessary to retrieve the result set. Without exactly showing the relationship between the hierarchy members, we show the skew of the data distribution over the first four product levels in Figure 8-17b. The distribution of the first two hierarchy levels is illustrated in Figure 8-18. The horizontal axis again displays the relationship of the members of the first two hierarchy levels.



(a) first hierarchy level



(b) first four hierarchy levels

Figure 8-17: Data distribution of the product dimension

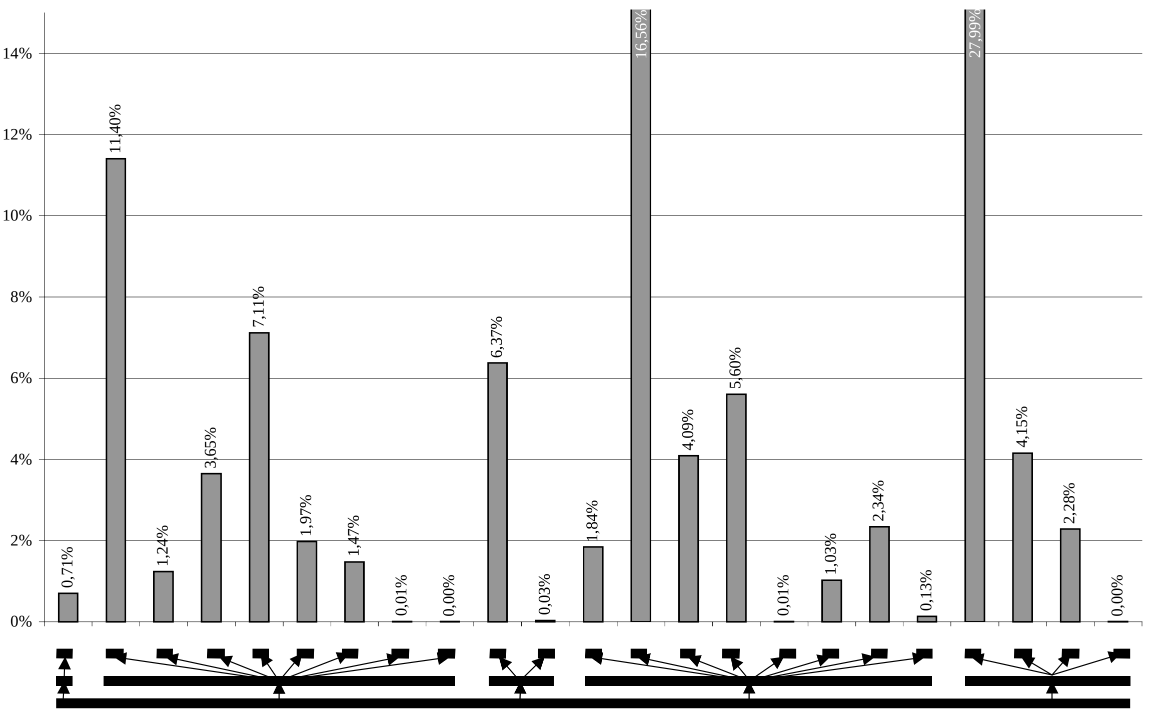


Figure 8-18: Data distribution of the first two hierarchy levels of the product dimension

8.3.2.3 The Customer Dimension

According to business administration literature 20% of the customers contribute to 80% of the business. The customer dimension of 'Juice & More' is a typical example for a classification of customers in such a company. A high number of customers (in this case 88%, the very left entry of Figure 8-19) are not classified (maybe they are not interesting for the company because of small turnover or it is not possible to find a classification). The classified customer groups contain 0% to 3% of the tuples stored in the table.[19] The hierarchical relationships on the horizontal axis show that the hierarchy of the customer dimension is not balanced, since several hierarchy members just have one child.

The consequence of the small number of classified customers is that in queries the customer dimension will be restricted to a small range (3%) and therefore the result set will be small.

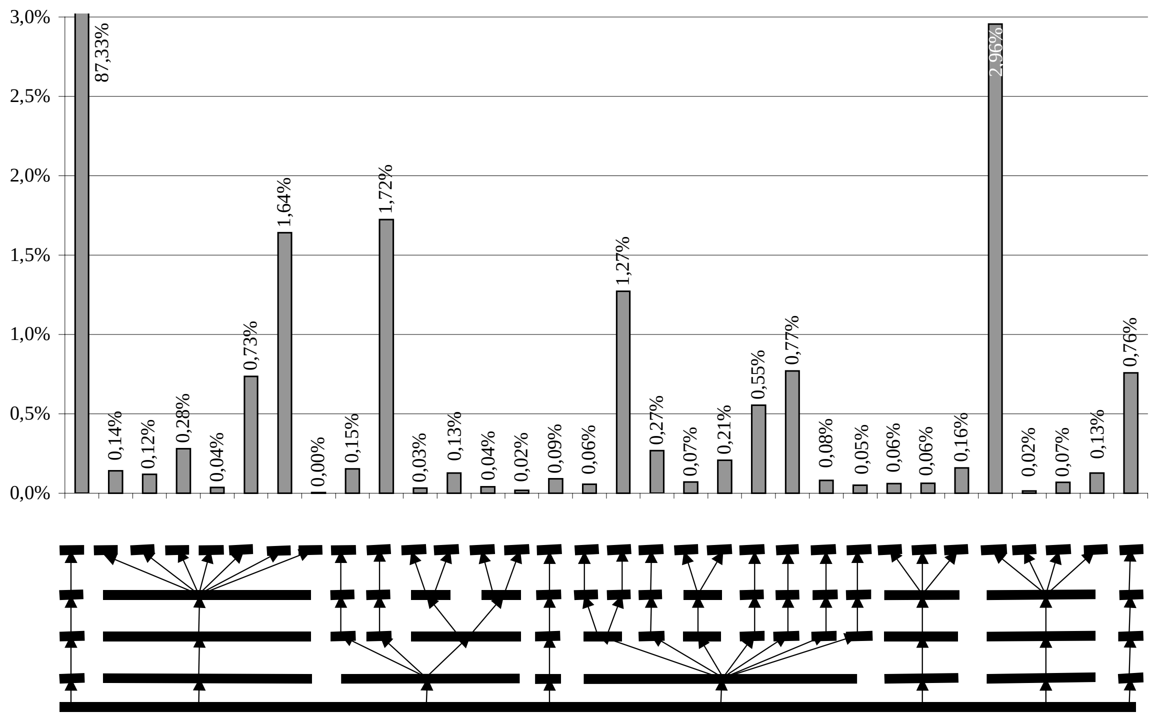
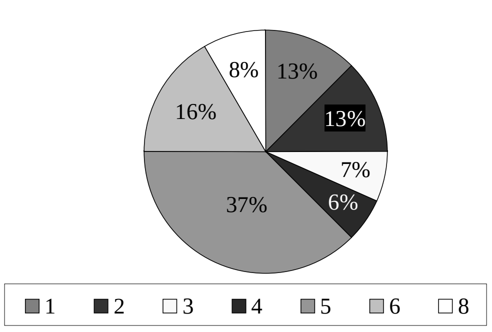


Figure 8-19: Data distribution of the customer dimension

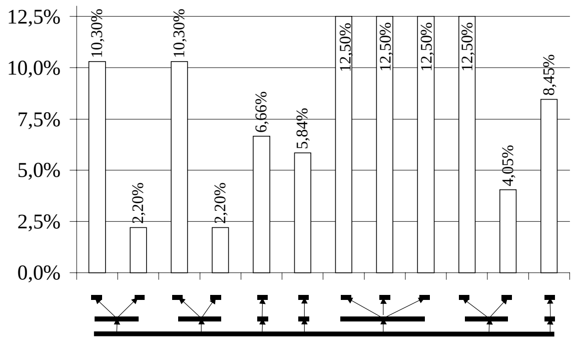
8.3.2.4 The Distribution Dimension

There are seven entries on the first level of the distribution hierarchy. The data distribution of the fact table with respect to the distribution dimension is highly skewed. Figure 8-20a shows the distribution of the first hierarchy level (i.e., sales organization), while Figure 8-20b shows the distribution of facts in the fact table for each distribution channel of each sales organization. Again the arrows indicate the hierarchical relationship of the members of neighboring hierarchy levels with the hierarchy root "All" at the bottom of the Figure.

[19] Note that the data in the 'Juice & More' warehouse is aggregated on a daily basis, thus the amount of data is usually compressed for large customers, thus the proportion of large customers is reduced.



(a)



(b)

Figure 8-20: Data distribution of the distribution dimension

8.3.3 Multidimensional Hierarchical Clustering of 'Juice & More'

Figure 8-21 shows the compound surrogates for the 'Juice & More' data warehouse, which are calculated as fixed length compound surrogates as described in Section 5.3.4. For any of the 4 hierarchies the length of the compound surrogate does not exceed 15 bits and thus can be stored in a single integer value. These compound surrogates are used as attributes for each of the four dimensions of 'Juice & More' to calculate the Z-address for each tuple of the 'Juice & More' fact table.

$$cs_{\text{product}} = \underbrace{p_{15}p_{14}p_{13}}_{\text{level 1}}\underbrace{p_{12}p_{11}p_{10}}_{\text{level 2}}\underbrace{p_9p_8p_7p_6p_5}_{\text{level 3}}\underbrace{p_4p_3p_2p_1}_{\text{level 4}}$$

$$cs_{\text{customer}} = \underbrace{c_{10}c_9c_8}_{\text{level 1}}\underbrace{c_7c_6c_5}_{\text{level 2}}\underbrace{c_4}_{\text{level 3}}\underbrace{c_3c_2c_1}_{\text{level 4}}$$

$$cs_{\text{time}} = \underbrace{t_6t_5}_{\text{level 1}}\underbrace{t_4t_3t_2t_1}_{\text{level 2}}$$

$$cs_{\text{distribution}} = \underbrace{d_5d_4d_3}_{\text{level 1}}\underbrace{d_2d_1}_{\text{level 2}}$$

Figure 8-21: Compound surrogates for each dimension of 'Juice & More'

The UB-Tree for the 'Juice & More' fact table consists of P = 878362 pages, which corresponds to:

$$l = \log_2 P = \log_2 878362 = 19{,}7$$

hierarchical split levels. With bit interleaving in the order of dimensions product, customer, time, and distribution the Z-address α for a tuple of the 'Juice & More' fact table is calculated as:

$$\alpha = \underbrace{p_{15}c_{10}t_6d_5p_{14}c_9t_5d_4p_{13}c_8t_4d_3p_{12}c_7t_3d_2p_{11}c_6t_2}_{\text{completely partitioned for any data distribution}}\ \underbrace{d_1}_{\text{partly split}}\ \underbrace{p_{10}c_5t_1p_9c_4p_8c_3p_7c_2p_6c_1p_5p_4p_3p_2p_1}_{\text{partitioned depending on the data distribution}}$$

The first 19 bits of the Z-address are guaranteed to be used to partition the four dimensional universe of the 'Juice & More' fact table. This means that the binary strings $p_{15}p_{14}p_{13}p_{12}p_{11}$ of the compound surrogate of product, $c_{10}c_9c_8c_7c_6$ of customer, $t_6t_5t_4t_3t_2$ of time and $d_5d_4d_3d_2$ of distribution are used to partition the universe. For each of the four dimensions the first hierarchy level is completely used for the partitioning. The second hierarchy level is used to a large extent to partition the universe. Therefore a restriction in the first hierarchy level will result in a reduction of the number of pages as determined by the data distribution of Section 8.3.2, i.e., a restriction of the product main group to "910" will reduce the number of pages to be retrieved to 27%, a restriction to "912" will result in a reduction to 6,41%. This holds for the restriction of the first hierarchy level in any dimension. If the top hierarchy level is restricted in several dimensions and the independence assumptions holds for these dimensions, the reduction is multiplicative. Table 8-5 shows the predicted selectivity calculated as the product of the selectivity in each dimension, the actual selectivity and the loaded pages in percent of the entire pages in the database for two queries, which restrict the first hierarchy level of three out of four dimensions.

$s_{customer}$	$s_{product}$	$s_{distribution}$	s_{time}	pages loaded	predicted selectivity	actual selectivity	loaded pages in %
0,78%	6,41%	37,5%	100%	178	0,0182%	0,0199%	0,0207%
87,34%	6,41%	37,5%	100%	18996	2,0981%	2,1618%	2,1627%

Table 8-5: Restrictions in the first hierarchy level in 3 of 4 dimensions

However, the data is not independently distributed in the entire 4-dimensional universe of 'Juice & More'. In this case the predicted selectivity does not describe the actual selectivity anymore. Thus some bits of the first 19 bits are correlated. This means that not all combinations of these bits occur and some partitioning will take places in the bits below bit number 19 of the Z-address. In this case the second level may already be completely partitioned and even a third hierarchy level partitioning may have started for some dimensions. A typical part of the multidimensional space where this will happen is the customer hierarchy "unclassified", which stores 87,34% of the customers. At most the three bits $c_{10}c_9c_8$ of the customer hierarchy are needed to distinguish these customers from all other customers. Thus for the unspecified customers the bits c_7c_6 of the first 19 bits of the Z-address are correlated to $c_{10}c_9c_8$ and two further bits may be used for partitioning. Thus d_1 will be split completely, p_{10} will be used for the partitioning and t_1 will be partly split (c_5 is also correlated to $c_{10}c_9c_8$). Actually, this is a puff-pastry effect, which due to the surrogate calculation is beneficial for query performance since it allows to have further partitioning steps for correlated hierarchy levels. Our measurements show that this effect also holds for other dimensions. Table 8-6 shows queries where the first two hierarchy levels of customer, product and time are restricted, whereas the distribution dimension is not restricted. The selectivity predicted by the cost functions here differs from the actual selectivity of the query because of dependencies in the data distribution. However, the percentage of pages loaded is similar to the actual selectivity of each query.

$s_{customer}$	$s_{product}$	$s_{distribution}$	s_{time}	pages loaded	predicted selectivity	actual selectivity	loaded pages in %
2,95%	3,64%	100%	38,41%	312	0,0414%	0,0310%	0,0355%
2,95%	27,99%	100%	38,41%	782	0,0318%	0,0851%	0,0890%

Table 8-6: Restriction in the first two hierarchy levels in 3 of 4 dimensions

Each of the 878362 pages of the 'Juice & More' fact table stores 30 tuples. All of the measurements showed that when restricting the first hierarchy level in each dimension in average 99,99% of the tuples on the pages contributed to the result set. A standard deviation of less than 0,001 for these measurements means that the multidimensional hierarchical clustering is perfect for multidimensional restrictions in the first hierarchy level. When additionally restricting the second hierarchy level in average 557 pages were loaded, where 10,7% of the tuples did not contribute to the result set. The standard deviation here was 0,06. Additionally restricting the third hierarchy level of each dimension usually created result sets with only one page.

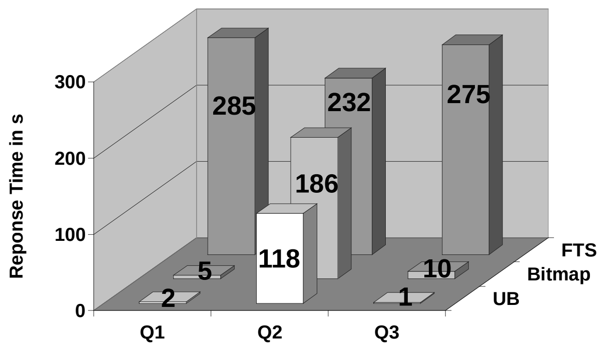
Thus multidimensional hierarchical restrictions are very well processed by UB-Trees storing compound surrogates which are created by the multidimensional hierarchical clustering technique introduced in Section 5.3.4. Again, the basic consideration about the UB-Tree in terms of dimensionality (cf. Section 3.9) and restricted dimensions (cf. Section 7.4.3) hold. For star schemas as used in present data warehousing applications this approach may significantly speed up query performance and reduce resource requirements in disk space and processing time. We are currently refining the technique of multidimensional hierarchical clustering by using hash clusters for the calculation of each individual surrogate. This refinement will be implemented to hierarchically cluster a data warehouse of marketing data provided by one of our project partners.

8.3.4 Performance Measurements

The measurements for 'Juice & More' were performed on a SUN Enterprise with four 300 MHz UltraSPARC processors and 2 GB RAM under Solaris 2.6. As secondary storage a partition on a SPARCstorage array with RAID level 0 (6 disks striping, 5-6 MB/s transfer rate per disk) was used. All measurements were done in a single-user environment.

It is important to note that our implementation still causes significant overhead due to the fact that we have implemented the UB-Tree on top of a DBMS and not in the kernel itself. First, the number of SQL statements that have to be processed (UB: 1 statement for each page in the result set, DBMS2 methods: 1 statement in total) leads to extensive inter-process communication (about 30% of the total processing time) and DBMS overhead (e.g., parsing of statements). Second, our table is larger than the one for the FTS and the bitmap indexes due to unimplemented compressing techniques in the UB-Tree (for 8 KB pages: UB: 878362 pages, FTS: 723539 pages, BII: FTS+31134 pages).

Figure 8-22 shows result set sizes and response times of the three example queries (Section 8.3.1). Q1 shows that the UB-Tree with multidimensional clustering is over 2 times faster than BII even for very small result sets. Q3 which is processed by the unoptimized UB-Tree at least 10 times faster than with any other access method undermines this observation.



Query	Loaded Tuples	Percentage of Database
Q1	8160	0,03%
Q2	1696416	6,52%
Q3	19752	0,08%

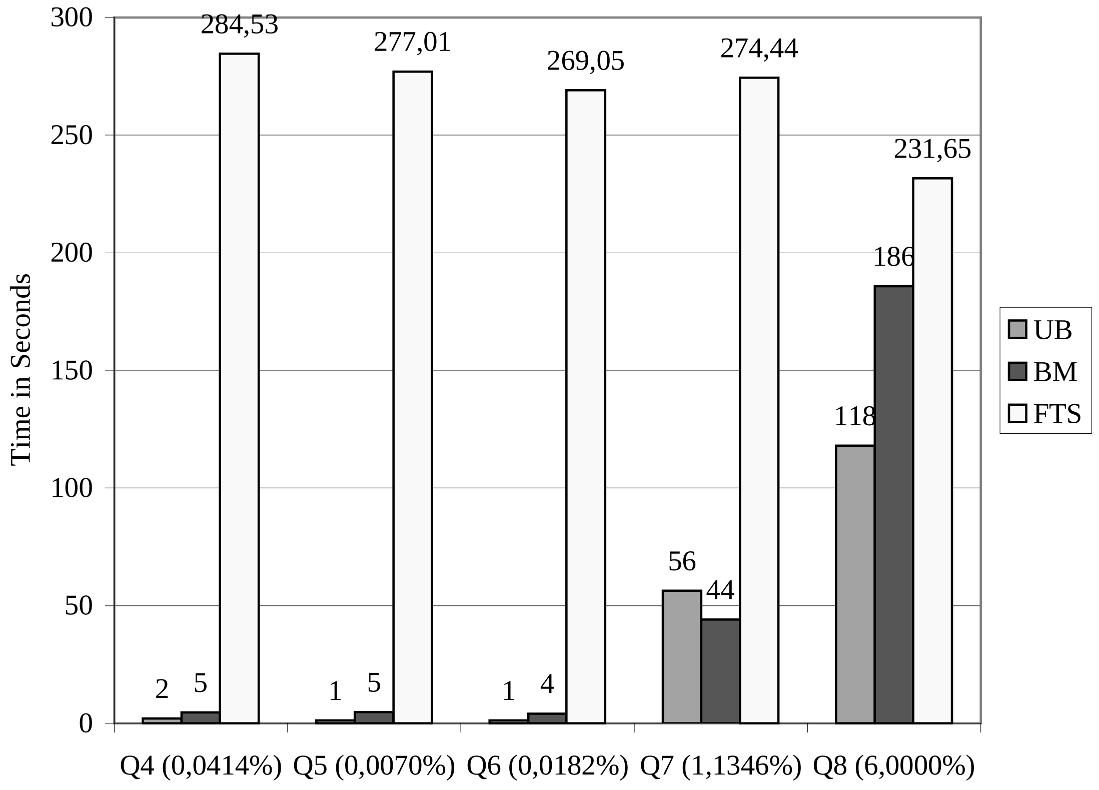
Figure 8-22: Query response times and result set sizes

The result set of query Q2 is quite large but the almost perfect clustering factor of the UB-Tree (in average more than 29 out of 30 tuples/page belong to the result set) still leads to a speed up of more than 30 % in comparison to BII. The time for FTS for Q2 differs from the times for Q1 and Q3 due to the less complex WHERE clause of the statement. The number of comparison operations is therefore much smaller than for the other queries which causes the faster execution.

All these results on real data show how well the multidimensional hierarchical clustering with UB-Trees works in practice and the accuracy of our theoretical cost model. In total more than 77% of all benchmark queries (28 out of 36) showed a speed up between a factor of 1.3 and 10 over traditional techniques [Pie98]. Figure 8-23 lists two instances for each of five further queries of that benchmark. For each query Table 8-7 lists the number of hierarchy levels that by each query are restricted to a point for each dimension. Since the data is non-uniformly distributed, the selectivity of each query depends on the exact point restriction, not only on the number of restricted hierarchy levels. We thus present two instances of the queries, each of which has a different selectivity.

	Customer	Product	Distribution	Time	Selectivity Instance 1	Selectivity Instance 2
Q4	2	2	0	1	0,0414%	0,3176%
Q5	1	1	1	1	0,0070%	3,4383%
Q6	1	1	1	0	0,0182%	2,0981%
Q7	1	0	0	1	1,1346%	1,1346%
Q8	0	1	0	1	6,0000%	34,0000%

Table 8-7: Restricted hierarchies and selectivities for five queries against the ‘Juice & More’ data warehouse



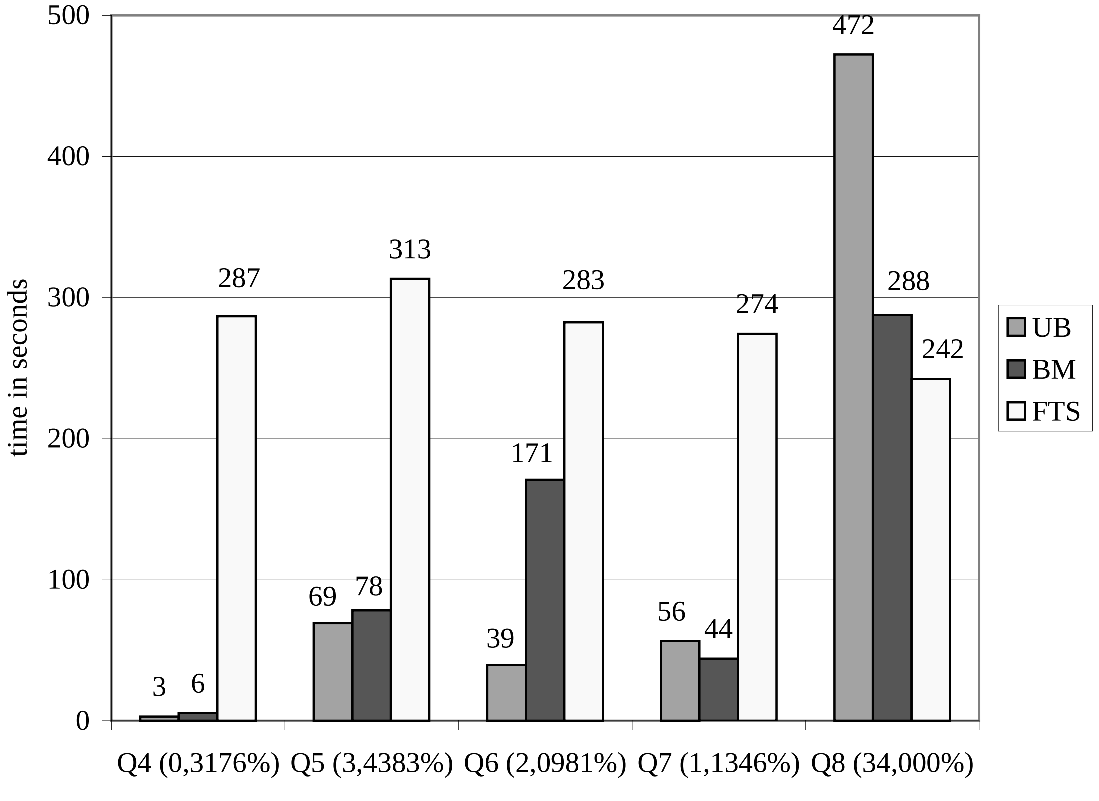


Figure 8-23: ‘Juice & More’ performance

8.3.5 Summary: Data Warehouse Performance Measurements

Our performance measurements have shown that multidimensional hierarchical clustering reduces the number of random accesses to the fact table for star joins and other queries with restrictions in multiple hierarchies. In addition, sort operations as necessary for grouping and aggregation are performed on the fly without additional I/O. For dimensionalities typical for data warehousing only I/O-time linear in size of the result set prior to aggregation and sublinear temporary storage are necessary to aggregate parts of a data cube. Thus secondary storage space and pre-computation time for many aggregates and bitmap indexes can be avoided. In addition the widely discussed view maintenance problem is minimized. The benchmark results for typical queries of a 7 GB real world retail data warehouse confirmed our analytical expectations and showed significant speedups up to a factor 10 in response time. Depending on the query, temporary storage requirements for sorting are reduced by several orders of magnitude. Our clustering approach also holds not only for ROLAP but also for MOLAP implementations of a data warehouse since both ROLAP fact tables and MOLAP data cubes can be clustered in this way.

A thousand things ... suddenly added up like a column of figures in her mind.

(William Humphrey)

Chapter 9

Summary

Before drawing conclusions and describing future research work, we briefly summarize the contribution of this thesis to the research in database management systems: We have analyzed the application of a multidimensional access method to RDBMS. After introduction of a formal model for multidimensionally partitioned relations we discussed several query types and identified the significance of multidimensional range queries and sort operations for query processing. Discussing current access methods we motivated the need for a multidimensional partitioning of relations. We described the UB-Tree in combination with algorithms for insertion, deletion, point queries, and range queries. We further introduced two new algorithms, the spiral algorithm for nearest neighbor queries with UB-Trees and the Tetris algorithm for efficient access to a table in arbitrary sort order. We also gave a detailed analysis of the UB-Tree space partitioning. We defined a cost model and compared the cost of range queries and sort operations for UB-Trees to the state of the art in current DBMS. The practical applicability of our approach was shown by performance measurements on a prototype implementation of UB-Trees on top of the RDBMS DBMS1 and DBMS2. These experiments with artificial data, for the TPC-D benchmark and for a star schema data warehouse confirmed the theoretically expected superiority of UB-Trees and the algorithms developed in this thesis over traditional access techniques with respect to both response times and storage requirements.

9.1 Conclusions

Next to a prototype implementation of most of the algorithms discussed in this thesis, our main contributions are the analytical cost model (Chapter 6), the Tetris algorithm (Section 4.7) and the technique of multidimensional hierarchical clustering (Section 5.3.4). The performance measurements of Chapters 7 and 8 proved, that all of these techniques are feasible and of practical relevance. Moreover, the expectations derived from our theoretical cost model in Chapter 6 are met by these performance measurements. In addition, some interesting algorithmic problems were solved during the work on this thesis (e.g., bit-interleaving for address calculation (cf. Section 5.3) or the reduction of the complexity of the calculation of the next Z-region intersection from exponential to linear (cf. Section 5.7.1).

The cost model may be used as a basis for cost-based query optimization. Using histograms [PIH+97] to determine the selectivity of the multidimensional restrictions, the selectivity based cost formula (Section 6.1.4) will also give a reasonable cost estimation for non-uniformly distributed partitioned multidimensional universes.

With enumeration types (cf. Section 5.3.3), multidimensional hierarchical clustering (cf. Section 5.3.4) and variable UB-Trees (cf. Section 5.3.5) we have introduced three techniques to overcome the puff-pastry effect which is inherent in Z-ordered data spaces (cf. Section 3.10). Suitable data modeling should ensure that these techniques are applicable in order to obtain a suitable multidimensional partitioning of a relation.

By the virtue of the cost model and by experiments we found that multidimensional indexes are useful for partitioning data according to up to 10 dimensions (cf. Section 3.9). For large dimensionalities a subset of the attributes should by chosen by proper physical data modeling techniques. However, dimensionalities of 6 – 10 dimensions are usual in relational data models, since one relation seldom has more than 6 – 10 foreign keys. Note that the number of key attributes is not identical to the dimensionality of the data space, which is often much smaller if foreign keys are composite keys.

The Tetris algorithm allows to accelerate most operations of relational query processing, if multidimensional range restrictions and/or sort operations are involved. Due to that fact one of our project partners has started to integrate UB-Trees and the Tetris algorithm into the kernel of its RDBMS. With suitable heuristics for query optimization, the Tetris algorithm thus may substantially speed up query processing in RDBMS. For dimensionalities typical for relational databases only I/O-time linear in the size of the result set and sublinear temporary storage are necessary to perform the Tetris algorithm. In contrast to a merge-sort algorithm, the sorted relation is produced in a continuous flow of operation. Therefore, using the Tetris algorithm, sorting is no longer a blocking operation. Thus internal pipelining is tremendously improved, since results are earlier transferred to other nodes of the operator tree or even to the user. Thereby the Tetris algorithm offers the chance to efficiently process iceberg queries for ranking [FSG+98], if the desired measure is used as a further dimension of the UB-Tree. The

Tetris algorithm then does not read the entire query box in the sorting dimension, but terminates after processing the first slices.

Aggregations can be calculated on-the-fly and allow better interactive response times. When sorting a relation or joining relations, restrictions in multiple attributes can be efficiently utilized in order to reduce I/O-cost and CPU-cost. The benchmark results for three queries of the TPC-D benchmark show speedups of up to two orders of magnitude in response time. Depending on the query, temporary storage requirements are reduced by several orders of magnitude. Further analysis indicates that our approach is useful for an even broader range of queries.

A star join with a point restriction of some hierarchy level of each dimension table results in a range restriction on each compound surrogate, if the fact table is organized with multidimensional hierarchical clustering using surrogate encoding for the foreign keys. In the same way intervals on the children of one hierarchy level result in a range of the corresponding compound surrogates. Thus, with multidimensional hierarchical clustering a star join on a schema with d dimensions creates a d-dimensional interval restriction on the fact table. Therefore star-joins in data warehouse benefit from the Tetris algorithm and the range query algorithm for UB-Trees as described in the previous paragraph. For dimensionalities typical for data warehousing only I/O-time linear in size of the result set prior to aggregation and sublinear temporary storage are necessary to aggregate parts of a data cube. Secondary storage space and pre-computation time for many aggregates and bitmap indexes can be avoided. In addition the widely discussed view maintenance problem is minimized. The benchmark results for typical queries of a 7 GB real world retail data warehouse confirmed our analytical expectations and showed significant speedups up to a factor 10 in response time. Depending on the query, temporary storage requirements for sorting are reduced by several orders of magnitude. Our clustering approach also holds not only for ROLAP but also for MOLAP implementations of a data warehouse since both ROLAP fact tables and MOLAP data cubes can be clustered in this way.

9.2 Future Work

In our future work we are particularly interested in a detailed study of relational query processing with multidimensional indexes. We are in the process of investigating a methodology for query optimization with multidimensional access methods, both for heuristics-based and cost-based query optimizers. We will do performance measurements in multi-user environments where we expect even more significant speedups. In a joint research project with TransAction Software we are currently integrating the UB-Tree into the DBMS1 kernel in order to reduce the overhead of the current implementation.

An interesting enhancement of UB-Trees might be the use of *Hilbert curves* [Hil91] as space filling curves for the UB-addresses. As our theoretical considerations in Section 3.3 showed, Hilbert curves will result in a better space partitioning because spatial proximity for both neighbors on the Hilbert curve is guaranteed. However, the algorithms for dealing with

Hilbert addresses (H-addresses) are more complicated than these for Z-addresses, since the subcube in space defined by each step of an H-address depends on the previous steps of the H-address. A prototype implementation and performance measurements of the overhead for H-address calculation and next intersection calculation of the range query algorithm will have to show if the better space partitioning pays off.

Query optimization with multidimensional access methods is an area which has not been researched in detail yet. The ability to handle multi-attribute restrictions with tuple clustered access methods will have to be taken into account by query optimizers. The Tetris operator of sections 4.7 and Chapter 8 shows that query optimization with multidimensional access methods will allow more complex operations to be handled by a single operator. In order to take advantage of multidimensional access methods, it is not sufficient for cost based optimizers to rely on one-dimensional statistical information. In addition, approaches like multidimensional histograms [PIH+96, Poo97] should be used here. For these applications our cost model of Chapter 6 could be further refined and adapted.

Multidimensional access methods can also be used for the organization of intermediate results. Very often a node of an operator tree gets an input stream sorted with respect to one attribute A_j, whereas a sorted processing with respect to another attribute A_k is required. A refinement of the Tetris algorithm could be used to perform a *sorted writing* of the intermediate result in sort order of A_k while taking advantage of the sorted input in order to reduce the cache size and to avoid the external sorting process. The implementation of this refinement of the Tetris algorithm for sorted writing as well as the implementation and analysis of the operators of the relational algebra with multidimensional access methods will be an interesting task of future research.

Another issue which has not been addressed in this thesis are the issue of *parallelization and data partitioning*. In our opinion many of the algorithms proposed in this thesis have a high potential for parallelization (e.g., the UB-Tree range query algorithm and the Tetris algorithm). Multidimensionally partitioning data with respect to several disks or RAID systems [PGK88] will also allow to exploit parallelism. Another issue with respect to data partitioning is the organization of huge databases on tertiary storage. Indexing of tertiary storage archives might also take our approach of multidimensional clustering into account.

Further research can also be done in the field of locking, where locking strategies might take advantage of the multidimensional organization of a relation. Spatial locking is a special kind of predicate locking which efficiently takes the physical organization of the relation into account in order to reduce contention problems. Page locking then corresponds to locking regions in multidimensional space. *Spatial locking* with UB-Trees might result in a hierarchical spatial locking concept which might be an improvement over classical locking mechanisms in both CPU-time for lock operations as well as space required for maintaining the locking information.

During the work for the thesis we also envisioned the lack of a *universal physical data model*. While a broad variety of data models has been introduced on the conceptual and logical level (e.g., E/R-model, relational model, object-oriented data models, multidimensional data models) no standardized physical data model has been established. Since for any logical model the data must be mapped to primary, secondary and tertiary storage, a universal physical data model would allow to use a single physical DBMS engine to implement any logical model. With appropriate mapping strategies a (time optimal, space optimal, etc.) physical model for a certain system configuration (logical DBMS schema, set of queries, set of access methods, hardware configuration) could be automatically derived (in analogy to the Auto Admin tool of Microsoft's SQL Server 7 [MS98a]). This might result in cost savings for DBMS application development, since a DBMS could offer views of the same physical data in several logical models. In addition, cost for DBMS administration might be reduced, since performance tuning is separated from the application data model and can be solely performed on the physical model.

Time present and time past are both perhaps present in time future. And time future contained in time past.

(T. S. Eliot)

References

[AFS93] R. Agrawal, C. Faloutsos, and A. Swami. *Efficient similarity search in sequence databases*. Proc. 4th Int. Conf. on Foundations of Data Organization and Algorithms, LNCS 730, 1993, pp. 69-84.

[AGM+90] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. *A Basic Local Alignment Search Tool*. Journal of Molecular Biology 215(3), 1990, pp. 403-410.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[AM90] D.J. Abel and D.M. Mark. *A comparative Analysis of some two-dimensional Orderings*. Intl. Journal Geographical Information Systems, 4(1), 1990, pp. 21-31.

[APB96] OLAP Council. *OLAP Council APB-1 Benchmark Information*. 1996.

URL: http://www.olapcouncil.org/bmark.html

[ARS97] K. Alsabti, S. Ranka, and V. Singh. *A One-Pass Algorithm for Accurately Estimating Quantiles for Disk-Resident Data*. VLDB, 1997, pp. 346-355.

[AS83] D.J. Abel and J.L. Smith. *A Data Structure and Algorithm based on a linear Key for a Rectangle Retrieval Problem*. Computer Vision 24, 1983, pp. 1-13.

[AS95] R. Agrawal and A.N. Swami. *A One-Pass Space-Efficient Algorithm for Finding Quantiles.* COMAD, 1995.

[Bau97] M. Bauer. *A Mass Loading Tool for UB-Trees*. Internship Report, FORWISS München, 1997.

[Bau98] M. Bauer. *Variable UB-Trees.* Master Thesis, TU München, 1998.

[Bay96] R. Bayer. *The universal B-Tree for multidimensional Indexing*. Technical Report TUM-I9637, Institut für Informatik, TU München, 1996.

[Bay97a] R. Bayer. *The universal B-Tree for multidimensional Indexing: General Concepts*. World-Wide Computing and Its Applications '97 (WWCA '97). Tsukuba, Japan, 10-11, Lecture Notes on Computer Science, Springer Verlag, March, 1997.

[Bay97b] R. Bayer. *UB-Trees and UB-Cache – A new Processing Paradigm for Database Systems.* Technical Report TUM-I9722, Institut für Informatik, TU München, 1997.

[BCE77] M.W. Blasgen, R.G. Casey, and K.P. Eswaran. *An Encoding Method for Multifield Sorting and Indexing*. Comm. of ACM 20(11), 1977, pp. 874-877.

[BDK92] F. Bancilon, C. Delobel, and P. Kanellakis (eds.). *Building an object-oriented Database System: The Story of O_2*. Addison-Wesley, 1992.

[Ben75] J.L. Bentley. *Multidimensional Binary Search Trees Used for Associative Searching.* Comm. of ACM 18(9), 1975, pp. 509-517.

[Ben79] J.L. Bentley. *Multidimensional Binary Search Trees in Database Applications.* IEEE TSE 5(4), 1979, pp. 333-340.

[BGW+81] P.A. Bernstein, N. Goodman, E. Wong, C.L. Reeve, and J.B. Rothnie. *Query Processing in a System for Distributed Databases (SSD-1).* ACM TODS 6(4), 1981, pp. 602-625.

[BK89] E. Bertino and W. Kim. *Indexing Technique for Queries on Nested Objects*. IEEE Transactions on Knowledge and Data Engineering, 1989, pp. 196-214.

[BKK96] S.D. Berchtold, D. Keim, and H.-P. Kriegel. *The X-Tree. An Index Structure for high-dimensional Data*. Proc. of 22nd VLDB Conf., 1996.

[BKK97] S. Berchtold, D. Keim, and H.-P. Kriegel. *Using Extended Feature Objects for Partial Similarity Retrieval*. VLDB Journal Vol. 6(4), 1997, pp. 333-348.

[BKS+90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. *The R*-Tree. An efficient and robust Access Method for Points and Rectangles*. Proc. of ACM SIGMOD Conf., 1990, pp. 322-331.

[BKS93] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. *Efficient Processing of Spatial Joins using R-Trees.* Proc. of ACM SIGMOD Conf., 1993, pp. 237-246.

[BM72] R. Bayer and E. McCreight. *Organization and Maintainance of large ordered Indexes.* Acta Informatica 1, 1972, pp. 173 – 189.

[BM98] R. Bayer and V. Markl. *The UB-Tree. Performance of Multidimensional Range Queries.* Technical Report TUM-I9814, Institut für Informatik, TU München, 1998.

[Böh98] C. Böhm. *Efficiently Indexing High-Dimensional Data Spaces*. Ph.D. Thesis, LMU München, 1998.

[BSH+98] M. Blaschka, C. Sapia, G. Höfling, and B. Dinter. *Finding Your Way through Multidimensional Data Models*. Proc. Intl. Workshop on Data Warehouse Design and OLAP Technology, Vienna, August 1998.

[BSW97] J. van den Bercken, B. Seeger, and P. Widmayer. *A General Approach to Bulk Loading Multidimensional Index Structures.* Proc. of 23rd VLDB Conf., Athens, Greece, 1997.

[BU77] R. Bayer and K. Unterauer. *Prefix B-Trees*. ACM TODS 2(1), 1977, pp. 11-26.

[CD97] S. Chaudhuri and U. Dayal. *An Overview of Data Warehousing and OLAP Technologies*. ACM SIGMOD Record 26(1), Marc 1997.

[CHH+91] J. Cheng, D. Haderle, R. Hedges, B.R. Iyer, T. Messinger, C. Mohan and Y. Wang. *An efficient hybrid hash join algorithm: a DB2 prototype*. Proc. of the 7th ICDE, Kobe, 1991, pp. 171-180.

[CI98] C. Chan and Y. Ioannidis. *Bitmap Index Design and Evaluation*. Proc. of ACM SIGMOD Conf., 1998.

[Cod70] E.F. Codd. *A Relational Model of Data for Large Shared Databanks.* Comm. of ACM 13(6), 1970, pp. 377-387.

[Com79] D. Comer. *The ubiquitous B-Tree*. ACM Computing Surveys, 11(2), 1979, pp. 121-138.

[CVS98] CVS, a version control system in the free software community. 1998.

URL: http://www.cyclic.com/

[Dat88] C.J. Date. *A Guide to the SQL Standard*,2nd edition. Addison-Wesley, 1988.

[Dev97] B. Devlin. *Data Warehouse from Architecture to Implementation*. Addision-Wesley Longman, Inc. 1997.

[DKO+85] D.J. DeWitt, R.H. Katz, F. Olken, L.D. Shapiro, M.R. Stonebraker, and D. Wood. *Implementation Techniques for Main Memory Database Systems*. Proc. of ACM SIGMOD Conf., 1984, pp. 1-8.

[Ell82] C. Ellis. *Extendible Hashing for Concurrent Operations and Distributed Data*. Proc. of ACM SIGMOD Conf. 1, 1982, pp. 106-115.

[Eva94] G. Evangelidis. *The hB$^{\pi}$-Tree: A Concurrent and Recoverable Multi-Attribute Index Structure*. Ph.D. thesis, North-Eastern University, Boston, MA, 1994.

[Fal85] C. Faloutsos. *Multi-attribute Hashing Using Gray Codes*. Proc. of ACM SIGMOD Conf., 1985, pp. 227-238.

[Fal88] C. Faloutsos. *Gray Codes for Partial Match and Range Queries*. IEEE TSE 14(10), 1988, pp. 1381-1393.

[FB74] R. Finkel and J.L. Bentley. *Quad-Trees: A Data Structure for Retrieval of Composite Keys.* Acta Informatica 4(1), 1974, pp. 1-9.

[Fen98] R. Fenk. *Design and Implementation of a UB-Tree Range Query Algorithm for a Set of Query Boxes*. Master Thesis, Technische Universität München, 1998.

[FG86] J.C. Freytag and N. Goodman. *Rule-Based Translation of Relational Queries into Iterative Programs*. Proc. of ACM SIGMOD Conf., 1986, pp. 206-214.

[FG89] J.C. Freytag and N. Goodman. *On the Translation of Relational Queries into Iterative Programs*. TODS 14(1), 1989, pp. 1-27.

[FG96] C. Faloutsos and V. Gaede. *Analysis of n-dimensional Quad-Trees using the Hausdorff Fractal Dimension*. Proc. of 22nd VLDB Conf., Bombay, India, 1996, pp. 40-50.

[FK94] C. Faloutsos and I. Kamel. *Beyond Uniformity and Independence: Analysis of R-Trees using the Concept of the Fractal Dimension*. Proc. of ACM SIGMOD-PODS Conf., Minneapolis, Minnesota, 1994, pp. 4-13.

[FNP+79] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. *Extendible Hashing – A Fast Access Method for Dynamic Files*. ACM TODS 4(3), 1979, pp. 315-344.

[FR89] C. Faloutsos and S. Roseman. *Fractals for Secondary Key Retrieval.* Proc. of 8th ACM SIGMOD-PODS Conf., 1989, pp. 247-252.

[Fre87] M. Freeston. *The BANG File: A new Kind of Grid File*. Proc. of ACM SIGMOD Conf., San Francisco, CA, 1987, pp. 260-269.

[Fre95] M. Freeston. A General Solution of the n-dimensional B-Tree Problem. Proc. of the ACM SIGMOD Conf., 1995, pp. 80-91

[Fre89] J.C. Freytag. *The Basic Principles of Query Optimization in Relational Database Management Systems*. IFIP, 1989, pp. 801-807

[Fri97] N. Frielinghaus: *Evaluierung der Einsatzfähigkeit des UB-Baums für das SAP-System R/3.* Master Thesis, TU München, 1997.

[FSG+98] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. Ullman. *Computing Iceberg Queries Efficiently*. Proc. of VLDB Conf., 1998, pp. 299-310.

[FSR87] C. Faloutsos, T. Sellis and N. Roussopoulos. *Analysis of Object-Oriented Spatial Access Methods*. Proc. of ACM SIGMOD Conf., 1987.

[Gae95] V. Gaede. *Optimal Redundancy in Spatial Database Systems*. Proc. SSD'95, Portland, LNCS Volume 951, 1995, pp. 96-116.

[Gar82] I. Gargantini. *An effective Way to Represent Quad-Trees*. Comm. of ACM 25(12), 1982, pp. 905-910.

[GBL+96] J. Gray, A. Bosworth, A. Layman, H. Pirahesh: *Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total.* Proc. of ICDE 1996, pp. 152-159

[GG97] V. Gaede and O. Günther. *Multidimensional Access Methods.* ACM Computing Surveys 30(2), 1997.

[GHR+97] H. Gupta, V. Harinarayan, A. Rajaraman, and D. Ullman. *Index Selection for OLAP*. Proc. of ICDE, 1997.

[GR97] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques. 3rd edition*. Morgan Kaufmann Publishers, 1997.

[Gra91] J. Gray. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann, San Mateo, CA, 1991.

[Gra93] G. Graefe. *Query Evaluation Techniques for Large Databases*. ACM Computing Surveys 25, 1993, pp. 73-170.

[Gre89] D. Greene. *An Implementation and Performance Analysis of Spatial Data Access Methods*. Proc. of 5th ICDE, 1989.

[Gün93] O. Günther. *Efficient Computations of Spatial Joins*. Proc. of 9th ICDE, Vienna, 1993.

[Gup97] H. Gupta. *Selection of Views to Materialize in a Data Warehouse*. Proc. of the Intl. Conference on Database Theory, Athens, Greece, January 1997.

[Gut84] A. Guttman. *R-Trees: A dynamic Index Structure for spatial Searching*. Proc. of ACM SIGMOD Conf., 1984, pp. 47-57.

[HAM+97] C.T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. *Range Queries in OLAP Data Cubes*. Proc. of ACM SIGMOD Conf., Tucson, Arizona, 1997, pp. 73-88.

[Hil91] D. Hilbert. *Über die stetige Abbildung einer Linie auf ein Flächenstück*. Math. Annln., 38, 1891, pp. 459-460.

[Hin85] K. Hinrichs. *Implementation of the Grid File: Design Concepts and Experience*. BIT 25, 1985, pp. 569-592

[HNK+90] L. Harada, M. Nakano, M. Kitsuregawa, and M. Takagi. *Query Processing Methods for Multi-Attribute Clustered Relations*. Proc. of 16th VLDB Conf., 1990, pp. 59-70.

[HR96] E.P. Harris and K. Ramamohanarao. *Join algorithm costs revisited*. VLDB Journal, 5, 1996.

[HSW88a] A. Hutflesz, H.-W. Six, and P. Widmayer. *Globally Order Preserving Multidimensional Linear Hashing*. Proc. of 4th ICDE, 1988, pp. 572-579.

[HSW88b] A. Hutflesz, H.-W. Six, and P. Widmayer. *Twin Grid Files: Space Optimizing Access Schemes*. Proc. of ACM SIGMOD Conf., 1988.

[HSW89] A. Hutflesz, H.-W. Six, and P. Widmayer. *The LSD-Tree: Spatial Access to Multidimensional Point and non-Point Objects*. Proc. of 15th VLDB Conf., Amsterdam, Netherlands, 1989, pp. 45-53.

[IBM97] IBM Corporation. *IBM DB2 Universal Database for UNIX Documentation*. IBM Corporation, 1997.

[Inf97] Informix Software Incorporation. *A New Generation of Decision Support Indexing for Enterprise-wide Data Warehouses*. 1997.

URL: http://www.informix.com/informix/corpinfo/zines/whitpprs/wpsps.pdf

[Inf98] Informix Corporation. *Informix Documentation: Dynamic Server 7.22 and Later.*, 1998.

[Inm96] W.H. Inmon. *Building the Data Warehouse*. John Wiley & Sons, Inc., 2nd edition, 1996.

[Jag89] H.V. Jagadish. *Incorporating Hierarchy in a Relational Model of Data*. Proc. of ACM SIGMOD Conf., Portland, Oregon, 1989.

[Jag90] H.V. Jagadish. *Linear Clustering of Objects with multiple Attributes*. Proc. of ACM SIGMOD Conf., 1990, pp. 332 – 342.

[Jag91] H.V. Jagadish. *A Retrieval Technique for Similar Shapes*. Proc. of ACM SIGMOD Conf., 1991, pp. 108-217.

[JC85] R. Jain and I. Chlamtac. *The P² Algorithm for Dynamic Calculation of Quantiiles and Histograms Without Storing Observations*. Comm. of ACM 28(10), 1985, pp. 1076-1085.

[JK84] M. Jarke and J. Koch. *Query Optimization in Database Systems*. ACM Computing Surveys 16(2), 1984, pp. 111-152.

[JL98] M. Jürgens and H.J. Lenz. *The R*a-Tree: An improved R*-Tree with Materialized Data for Supporting Range Queries on OLAP-Data*. DWDOT Workshop, Vienna, 1998.

[KE97] A. Kemper, A. Eickler. *Datenbanksysteme : Eine Einführung.* 2. aktual. u. erw. Aufl., Oldenbourg Verlag , München, 1997, p. 504

[KF94] I. Kamel and C. Faloutsos. *Hilbert R-Tree: An Improved R-Tree using Fractals*. Proc. of 20th VLDB Conf., 1994, pp. 500-509.

[Kim96] R. Kimball. *The Data Warehouse Toolkit : PRACTICAL TECHNIQUES FOR BUILDING DIMENSIONAL DATA WAREHOUSES.* JOHN WILEY & SONS, INC., New York, 1996, p. 388.

[Kim96] R. Kimball. *The Data Warehouse Toolkit*. John Wiley & Sons, New York. 1996.

[KKD89] W. Kim, C. Kim, and A. Dale. *Indexing techniques for object-oriented database*. In Object-Oriented Concepts, Databases and Applications. Addison-Wesley, 1989, pp. 371-394.

[Knu68] D.E. Knuth. *The Art of Computer Programming Volume 1: Fundamental Algorithms*. Addison-Wesley, Reading, MA, 1968.

[Knu73] D.E. Knuth. *The Art of Computer Programming Volume 3: Sorting and Searching*. Addison-Wesley, Reading, MA, 1973.

[KP88] M.H. Kim and S. Pramanik. *Optimal File Distribution for Partial Match Retrieval.* Proc. of ACM SIGMOD Conf., Boston, NA, 1984, pp. 186-196.

[Kri84] H.-P. Kriegel. *Performance Comparison of Index Structures for Multi-Key Retrieval*. Proc. of ACM SIGMOD Conf., Boston, MA, 1984, pp. 186-196.

[KS87] H.-P. Kriegel and B. Seeger. *Multidimensional Dynamic Quantile Hashing is Very Efficient for Non-Uniform Record Distributions*. ICDE, pp. 10-17, 1987.

[KSF+96] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. *Fast Nearest Neighbor Search in Medical Image Databases*. Proc. of 22nd VLDB Conf., Bombay, India, 1996, pp. 215-226.

[LS90] D. Lomet and B. Salzberg. *The hB-Tree: A Multiattribute Indexing Method with good guaranteed Performance*. ACM TODS, 15(4), 1990, pp. 625 – 658.

[Lum70] Y.V. Lum. *Multi-Attribute Retrieval with Combined indexes*. Comm. of ACM 13(11), November 1970, pp. 660-665.

[MB97a] V. Markl. *The Tetris-Algorithm for multidimensional Sorted Reading from UB-Trees*. Internal Report, FORWISS München, 1997.

[MB97b] V. Markl and R. Bayer. *A Cost Model for multidimensional Queries in Relational Database Systems*. Internal Report, FORWISS München, 1997.

[MB97c] V. Markl and R. Bayer. *Variable UB-Trees for the efficient Indexing of arbitrarily distributed multidimensional Data*. Internal Report, FORWISS München, 1997.

[MB98] V. Markl and R. Bayer. *The Tetris-Algorithm for Sorted Reading from UB-Trees*. In “Grundlagen von Datenbanken”, 10th GI Workshop, Konstanz, 1998.

[McG96] F. McGuff. *Data Modeling Patterns for Data Warehouses*. Comprehensive Systems Inc., July 1996.

[ME92] P. Mishra and M.H. Eich. *Join Processing in Relational Databases*. ACM Computing Surveys, Vol. 24 No.1, 1992, pp. 194-211.

[Mer81] T.H. Merret. *Why Sort-Merge gives the best Implementation of the Natural Join.* ACM SIGMOD Record 13(2), 1981, pp. 39-51.

[MHW+90] C. Mohan, D. Haderle, Y. Wang, and J. Cheng. *Single Table Access Using Multiple Indexes: Optimization, Execution and Concurrency Control Techniques.* International Conference on Extending Database Technology, 1990, pp. 29-43.

[MOD99] *MISTRAL Online Documentation*. TU München, 1999.

URL: http://mistral.informatik.tu-muenchen.de

[Moe98] G. Moerkotte. *Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing*. Proc. of 24th VLDB Conf., New York, USA, 1998.

[MS98a] Microsoft Corporation. *SQL Server 7.0*. 1998.
URL: http://www.microsoft.com/sql/

[MS98b] Microsoft Corporation. *OLE DB for OLAP Programmer's Reference*. February 1998.
URL: http://www.microsoft.com/data/oledb/olap

[Mul71] J.K. Mullin. *Retrieval-Update Speed Tradeoffs Using Combined Indexes*. Comm. of ACM 14(12), Dec. 1971, pp. 775-776.

[MZB99] V. Markl, M. Zirkel, and R. Bayer. *Processing Operations with Restrictions in Relational Database Management Systems without external Sorting*. Proc. of ICDE, Sydney, Australia, 1999.

[NHS84] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. *The Grid-File*. ACM TODS, 9(1), March 1984, pp. 38-71.

[OLA98] The OLAP Council. *MDAPI TM The OLAP Application Program Interface Version 2.0 Specification*. January 1998.

[OM84] J. A. Orenstein and T.H. Merret. *A Class of Data Structures for Associate Searching*. Proc. of ACM SIGMOD-PODS Conf., Portland, Oregon, 1984, pp. 294-305.

[OQ97] P. O´Neill and D. Quass. *Improved Query Performance with Variant Indexes*. Proc. of ACM SIGMOD Conf., Tucson, Arizona,1997, pp. 38-49.

[Ora97] Oracle Corporation. *Oracle 8 Documentation*. Oracle Corporation, 1997.

[Ore89] J. Orenstein. *Redundancy in spatial Databases*. Proc. of ACM SIGMOD Conf., 1989, pp. 294-305.

[Ova99] D. Ovadya. *Porting the UB-Tree to DB2*. Internship Report, FORWISS München, 1999.

[PDF+98] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. *The Multidimensional Database System RasDaMan*. Proc. of ACM SIGMOD Conf., 1998, pp. 575-577.

[Pfa99] M. Pfadenhauer. *Porting the UB-Tree to Informix Universal Server*. Internship Report, FORWISS München, 1999.

[PGK88] D.A. Patterson, G.A. Gibson, and R.H. Katz. *A Case for Redundant Arrays of Inexpensive Disks (RAID)*. Proc. of ACM SIGMOD Conf., 1988, pp. 109-116.

[PH90] D.A. Patterson and J.L. Hennessy. *Computer architecture: a quantitative approach*. Kaufmann, San Mateo, Calif., 1990.

[Pie97] R. Pieringer. *Porting the UB-Tree to Oracle*. Internship Report, TU München, 1997.

[Pie98] R. Pieringer. *Evaluation of the UB-Tree in the SAP Environment*. Master Thesis, TU München, 1998.

[PIH+96] V. Poosala, Y.E. Ioannidis, P.J. Haas, and E.J. Shekita. *Improved Histograms for Selectivity Estimation of Range Predicates*. Proc. of ACM SIGMOD Conf., 1996, pp. 294-305.

[Poo97] V. Poosala. *Histogram-Based Estimation Techniques in Database Systems*. Ph.D thesis, Univ. of Wisconsin-Madison, 1997.

[PS85] F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.

[PST+93] B.U. Pagel, H.W. Six, H. Toben, and P. Widmayer. *Towards an Analyis of Range Query Performance in Spatial Data Structures*. Proc. of 12th ACM SIGMOD-PODS Conf., Washington D.C., 1993, pp. 214-221.

[Red97] Redbrick Systems. *Star Schema processing for Complex Queries*. 1997.
URL: http://www.redbrick.com/rbsg/whitepapers/starJoin.pdf

[Rit89] G. Ritter. *Information Processing 89,* Proc. of IFIP, World Computer Congress, San Francisco, USA, August 28 - September 1, 1989.

[Rob81] J.T. Robinson. *The K-D-B-Tree: A Search Structure for large multidimensional dynamic indexes*. Proc. of ACM SIGMOD Conf., 1981, pp. 10-18.

[Rot91] D. Rotem. *Spatial Join Indices*. Proc. of ICDE, 1991, pp. 500-509.

[Sag94] H. Sagan. *Space Filling Curves*. Springer Verlag, Berlin/Heidelberg/New York, 1994.

[Sal88] B. Salzberg. *File Structures: An Analytic Approach*. Prentice Hall, 1988.

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison Wesley, 1990

[SAP99] SAP AG. *SAP Homepage*. 1999
URL: http://www.sap.com/

[Sar97] S. Sarawagi. *Indexing OLAP data*. Data Engineering Bulletin 20 (1), 1997, pp. 36-43.

[Sch98] M. Schramm. *UB-Tree Insertion and Deletion Functions.* Internship Report, TU München, 1998.

[SDN+96] A. Shukla, P. Deshpande, J. Naughton, and K. Ramasamy. *Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies.* Proc. of 22nd VLDB Conf., Mumbai (Bombay), India, 1996.

[SDN+98] A. Shukla, P. Deshpande, and J. Naughton. *Materialized View Selection for Multidimensional Datasets*. Proc. of ACM SIGMOD Conf., 1998.

[SH94] H. Shawney and J. Hafner. *Efficient Color Histogram Indexing*. Proc. Int. Conf. on Image Processing, 1994, pp. 66-70.

[SK90] B. Seeger and H.-P. Kriegel. The Buddy Tree: *An Efficient and Robust Access Methods for Spatial Database Systems.* Proc. of 14th VLDB Conf., 1988, pp. 360-371

[Spe91] G. Specht. *LOLA, LDL, NAIL!, RDL, ADITI and STARBURST: A Comparison of Deductive Database Systems*. Institut für Informatik. TU-München 1991.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. *The R+-Tree: A Dynamic Index for Multi-Dimensional Objects.* Proc. of 13th VLDB Conf., Brighton, England, 1987, pp. 507-518.

[Sto94] M. Stonebraker (ed.). *Readings in Database Systems 2nd edition*. Morgan Kaufmann, 1994.

[Str99] M. Streichsbier. *Implementation and Analysis of the Spiral Algorithm for Nearest-Neighbor Queries with UB-Trees.* Master Thesis, TU München, 1999.

[Tam82] M. Tamminen. *The extendible cell method for closest point problems*. BIT 22, 1982, pp. 27-42

[TAS98] TransAction Software GmbH. *TransBase Relational Database System Version 4.3, Manual.* Transaction Software GmbH, München, Germany, 1998.

[TH81] H. Tropf and H. Herzog. *Multidimensional Range Search in Dynamically Balanced Trees.* Angewante Informatik, 2/1981.

[TPC97] Transaction Processing Performance Council. *TPC Benchmark D (Decision Support)*. Standard Specification, Revision 1.2.3. June 1997.
URL: http://www.tpc.org

[Ull88] J.D. Ullman. *Database and Knowledge Based Systems Volume I.* Computer Science Press, Rockville, MD, 1988.

[Ull89] J.D. Ullman. *Database and Knowledge Based Systems Volume II.* Computer Science Press, Rockville, MD, 1989

[WB97] M.C. Wu and A.P. Buchmann. *Research Issues in Data Warehousing*. BTW'97. 1997.

[WB98] M.C. Wu and A.P. Buchmann. *Encoded Bitmap Indexing for Data Warehouses.* Proc. of ICDE, Orlando, 1998.

[Wid95] J. Widom. *Research Problems in Data Warehousing*. Proc. of 4th CIKM, November 1995.

[WSB98] R. Weber, H.J. Schek, and S. Blott. *A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces*. Proc. of VLDB Conf., New York, 1998

[WZ98] R. Wunderling and M. Zöckler. *DOC++, a free source code documentation tool*. 1998.

URL: http://www.zib.de/Visual/software/doc++/

[ZDN+98] Y. Zhao, R. Deshpande, J. Naughton, and A. Shukla. *Silmutaneous Optimization and Evaluation of Multiple Dimensional Queries*. Proc. of ACM SIGMOD Conf., 1998.

[ZSL98] C. Zou, B. Salzberg, and R. Ladin. *Back to the Future: Dynamic Hierarchical Clustering.* Proc. of ICDE, 1998, pp. 578-587.

[Zwi96] D. Zwillinger (ed.). *Standard Mathematical Tables and Formulae*. 30th edition, CRC Press, 1996.

List of Figures

List of Tables

List of Definitions

List of Theorems and Lemmas

Index

Term	**Description**	**Symbol**	**see also**
area, geometric	⇒volume		Section 2.1.3
arity	number of attributes of a relation	*d*, *d'*, d_R	Section 2.1
BII	bitmap index intersection		Section 2.4.2, Section 8.3.4, Section 6.3
bit interleaving	efficient implementation of the Z-address calculation		Section 5.3.1
C-address	index of a point on the compound curve		Definition 3-2
cardinality	number of elements in a set		Section 2.1.1
C-curve	synonym to ⇒compound curve		
clustering	data that is likely to be accessed together is placed physically close to each other		Section 2.3.2
clustering, page	pages are stored in index order on physical storage		Definition 2-23, Section 2.3.2.3, Section 6.2
clustering, tuple	tuples are stored in index order on a page		Definition 2-23, Section 2.3.2.2, Section 6.2
column	synonym to ⇒attribute		Section 2.1

compound curve	space filling curve created by concatenating the co-ordinates of the points in some fixed order		Definition 3-4
compound ordering	linear ordering of the multidimensional space created by the C-curve	$\lessdot$	Section 2.3.2, Example 2-6
coordinate	value of a point in one dimension	x_i, y_i, z_i	Section 2.1
correspondence between pages and regions		$\leftrightarrow$	Definition 2-16
CSI	composite secondary index (see also ⇒IOT)		Section 7.4.2, Section 6.3, Section 2.3
C-value	synonym to ⇒C-address		
dimension			Section 2.1
dimensionality	synonym to ⇒arity		Section 2.1
distance			Section 2.1.3
domain		$\mathbb{D}$, $\mathbb{D}_i$, Ω_i	Section 2.1.1
domain, actual	subset of a domain actually used in a relation	$\mathbb{V}_i$	Definition 3-34
domain, maximum value		υ_i	Section 2.1.1
domain, minimum value		λ_i	Section 2.1.1
domain, multidimensional	cross product of one-dimensional domains	Ω	Section 2.1.1
FTS	full table scan		Section 2.3, Section 2.4, Section 7.4.2, Section 6.3

H-address	index of a point on the Hilbert curve		Section 3.1
H-curve	synonym to ⇒Hilbert curve		
Hilbert curve	space filling curve created by the leitmotif of D. Hilbert [Hil91]		Section 3.1
H-value	synonym to ⇒H-address		
idealized uniform partitioning	special region distribution of a multidimensionally partitioned universe		Section 3.7.3, Section 6.1
index candidate	attribute restricted or sorted during the processing of a query		Definition 2-22
interval, multidimensional		$[[x, y]]$, $]]x, y]]$, $[[x, y[[$, $]]x, y[[$	Definition 2-8
interval, one-dimensional		$[a, b]$, $]a, b]$, $[a, b[$, $]a, b[$	Definition 2-7
IOT	index organized table, clustering B^*-Tree		Section 7.4.2, Section 2.4.1
Lebesgue curve	synonym to ⇒Z-curve		
length	⇒volume	vol	Section 2.1.3
lexicographic ordering	synonym to ⇒compound ordering	$<$	Section 2.3.2, Example 2-6
MSI	multiple secondary index intersection (see also ⇒BII)		Section 2.4.2, Section 7.4.2, Section 6.3
Multidimensional Hierarchical Clustering	encoding technique for multidimensional hierarchies		Section 5.3.4

neighbor of a point	point, which only differs in one coordinate from a given point and the difference between the two points in this coordinate is the limit of resolution		Section 2.1.1
normalization, scalar		$\hat{a}$	Definition 2-5
normalization, tuple		^	Definition 2-6
page	byte container or (ordered) set storing tuples of a relation		Definition 2-15
point, d-dimensional	vector of coordinates which defines a location in d-dimensional space	x, y, z	Section 2.1
predecessor, point			Definition 3-9
predecessor, value			Definition 3-8
puff pastry degree	degree of unsymmetry of a Z-ordered space		Section 3.10
query			Definition 2-18
query box		Q	Definition 2-21
query, arbitrary volume			Section 2.2.3
query, exact match			Section 2.2.1
query, partial match			Section 2.2.1
query, partial range			Section 2.2.2

query, range			Section 2.2.2
region	subspace of Ω		Definition 2-14
relation	set of tuples	R, S	Section 2.1
relation, base space	⇒multidimensional domain of a relation		Section 2.1.1
relation, multidimensionally partitioned	set of tuples stored on a set of pages, where each page corresponds to a region in multidimensional space	P_R, P_S	Section 2.1.5, Definition 2-17
relation, partitioned	set of tuples stored on a set of pages	P_R, P_S	Section 2.1.5, Definition 2-17
restriction	logical predicate defined by a query		Definition 2-20
restriction interval	subspace defined by a query		Definition 2-21
result attribute	attribut projected into the result set of a query		Definition 2-22
result set	tuples satisfying a query condition		Definition 2-18
row	synonym to ⇒tuple		Section 2.1
selectivity	percentage of tuples in the database satisfying a restriction		Definition 2-19
spiral algorithm	algorithm to process nearest neighbor queries on multidimensionally partitioned data spaces		Section 4.6
SSI	single secondary index		Section 6.3, Section 2.3

successor, point			Definition 3-9
successor, value			Definition 3-8
table	synonym to ⇒relation	*R*, *S*	Section 2.1
Tetris algorithm	algorithm to process sort operations while processing multi-attribute restriction on multidimensionally partitioned data spaces		Section 4.7, Section 5.3.4.6, Section 6.4
tuple, d-dimensional		*x*, *y*, *z*	Section 2.1.1
type	type of an attribute		Section 2.1
UB-Tree	multidimensional access method based on Z-ordering		Definition 4-1
variable UB-Tree	encoding technique for independent, arbitrarily distributed dimensions		Section 5.3.5
volume	percentage of space covered by a subspace compared to the entire multidimensional space	vol	section 2.1.3
Z-address	index of a point on the Z-curve		Definition 3-3, Definition 3-19, Definition 3-20
Z-area	subspace of the multidimensional space constructed by a Z-address	Λ_1, Λ_2, Λ_3, ...	Definition 3-19
Z-curve	space filling curve created by bit-interleaving of the co-ordinates of the points		Definition 3-4

Z-ordering	linear ordering of multidimensional space created by the Z-curve	$\prec$	Definition 3-5
Z-region (closed)	subspace covered by the part of the Z-curve starting with Z-address α and ending with Z-address β	$[\alpha : \beta]$	Definition 3-17, Section 3.6
Z-region (open)	subspace covered by the part of the Z-curve starting with Z-address α and ending with Z-address β, where α is not included in the region	$]\alpha : \beta]$	Definition 3-23
Z-value	synonym to ⇒Z-address		