INSTITUT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Advanced Concepts and Applications of the UB-Tree

Robert Josef Widhopf-Fenk

Institut für Informatik
der Technischen Universität München

Advanced Concepts and Applications of the UB-Tree

Robert Josef Widhopf-Fenk

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

Doktors der Naturwissenschaften

genehmigten Dissertation.

Vorsitzende:		Univ.-Prof. G. J. Klinker, Ph.D.
Prüfer der Dissertation:	1.	Univ.-Prof. R. Bayer, Ph.D., emeritiert
	2.	Univ.-Prof. A. Kemper, Ph.D.

Die Dissertation wurde am 05.08.2004 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 07.02.2005 angenommen.

Acknowledgements

First of all, I want to thank my supervisor Rudolf Bayer for his advice, guidance, support, and patience. Without his invitation to be a Ph.D. student, I would have never been working on database research.

I owe my former colleagues in the MISTRAL project many thanks for fruitful discussions, proofreading, inspirations, and motivating me: Volker Markl, Frank Ramsak, Martin Zirkel and our “external” member Roland Pieringer.

I like to thank my students, Oliver Nickel, Chris Hodges and Werner Unterhofer contributing to this work by supporting the implementation of algorithms and performing experiments.

Special thanks to our project partners at TransAction, Teijin, GfK, and the members of the EDITH project.

And last but not least, many thanks to Michael Bauer, Bernd Reiner, and Charlie Hahn for discussions, distraction and small talk. Specifically: to Charlie for the ice cream; to Alex S. for the Gauloise; to Evi Kollmann for all your help, even when you were loaded with other work you always were at hand. Also thanks to everyone I accidently forgot.

To my love Birgit
for your patience,
for our children Matilda and Valentin,
and for going through this with me!

Abstract

The UB-Tree is an index structure for multidimensional point data. By name, it claims to be universal, but this imposes a huge burden, as there are few things which really prove to be universal. This thesis takes a closer look at aspects where the UB-Tree is not *universal* at a first glance.

The first aspect is the discussion of space filling curves (SFC), in particular comparing the Z-curve and the Hilbert-curve. The Z-curve is used to cluster data indexed by the UB-Tree and we highlight its advantages in comparison to other SFCs. While the Hilbert-curve provides better clustering, the Z-curve is superior w.r. to other metrics, i.e., it is significantly more efficient to calculate addresses and the mapping of queries to SFC-segments, and it is able to space efficiently index arbitrary universes. Thus the Z-curve is more *universal* here.

The second aspect are bulk operations on UB-Trees. Especially for data warehousing the bulk insertion and deletion are crucial operations. We present efficient algorithms for incremental insertion and deletion.

The third aspect is the comparison of the UB-Tree with bitmap indexes used for an example data warehousing application. We show how performance of bitmap indexes is increased by clustering the base table according to a SFC. Still the UB-Tree proves to be superior.

The fourth aspect is the efficient management of data with skewed data distributions. The UB-Tree adapts its partitioning to the actual data distribution, but in comparison to the R-Tree, it suffers from being not able to prune search path leading to unpopulated space (= dead space). This is caused by partitioning the complete universe with separators. We present a novel index structure, the *bounding UB-Tree* (BUB-Tree), which is a variant of the UB-Tree inheriting its worst case guarantees for the basic operations while efficiently addressing queries on dead space. In comparison to R-Trees, its query performance is similar while offering superior maintenance performance and logarithmic worst case guarantees, thus being more universal than the R*-Tree.

The last aspect addressed in this thesis is the management of spatial data. The UB-Tree is an index designed for point data, however also spatial objects can be indexed efficiently with it by mapping them to higher dimensional points. We discuss different mapping methods and their performance in comparison to the RI-Tree and R*-Tree.

Our conclusion: The UB-Tree comes closer to being an universal index than any other competing index structure. It is flexible, dynamic, relatively easy to integrate into a DBMS kernel, and provides logarithmic worst case guarantees for the basic operations of insertion, deletion, and update. By extending its concepts to the BUB-Tree it is also able to efficiently support skewed queries on skewed data distributions.

Contents

List of Figures

List of Tables

Preface

Every thesis should address three topics: the objective, the problem and the solution.

The Objective:
Getting answers within guaranteed and acceptable time is what we strive for.

The Problem:
The management of data is non trivial with real world data, due to its large quantities and skewed data distributions.

The Solution:
Managing data by an efficient and flexible indexing technique is the solution to reach the objective.

While anything might be encoded as a sequence of bits, the limited memory of todays computers and efficient processing support by CPUs imposes restrictions. Therefore, all basic data types (e.g., unsigned, integer, and floating point numbers) directly supported by the CPU are restricted to a given domain usually defined by the architecture of the CPU, i.e., a maximum of 32 bits is used on most systems resulting in a domain of $[0, 2^{32-1}]$ for unsigned integers. More complex data can be stored by composing structures from the basic data types.

Chapter 1

Introduction

This chapter begins with an informal introduction to the objective we are addressing in this thesis. We discuss the problems and then focus on the solutions.

1.1 The Objective: Getting Information Fast

The answer to the great question of life, the universe and everything is 42. [Ada80]

We are always seeking for new knowledge in order to understand information and so we start asking questions. Nowadays, vast amounts of information are generated, but this information is useless without creating new knowledge from it.

To achieve this, we develop models and we fill them with data. The objective of these models is to understand the world, gain new knowledge and to allow to project future events. But, the real world is too complex for us to be described by just one generic model. Therefore, we create simplified models for subsets of the real world that may give us answers to questions specific to this subset of the real world.

Computers allow us to create models more complex than ever before, but we are far away from our objective: one comprehensive model giving us all the answers we are looking for and including all the information we have. And even creating a model is just half the work, since efficiently implementing it to give answers within a reasonable time frame is another challenging issue.

Queries asked again and again can be optimized as their properties are known in advance. However, new data and knowledge almost certainly triggers different questions. Specifically, interesting questions are usually ad-hoc questions, i.e., we do not know them in advance. Thus it is not possible to prepare an answer or fully optimize processing of them. Depending on the complexity of our question we may wait for a certain time, but still they should be answered within an acceptable time horizon.

1.2 The Problem: The Properties of Real World Data

In the following we will discuss the properties of the data that should be supported. In general, data can be classified into two basic categories by its source:

Artificial Data: data generated by a given algorithm or function.

Real World Data: data gathered from "unpredictable" processes.

Most real world data is information gathered from events, e.g., sales, phone calls, bank transactions, movement of mobile phones, product sales etc.. This data is usually confidential and will not be shared. Artificial data can emulate real world data without compromising privacy and this can be utilized to create standardized benchmarks [Tra99], while avoiding to share real data.

Real world data is predictable to some extent, but things may instantly and unexpectedly change. Consider information about phone calls, i.e., the caller, callee, time and duration of call. Most private calls will be in the evening hours and on week ends, while business calls take place during the working hours. While these events are likely to repeat, an exact prediction of calls is not possible. Changes in the location, number or partners make this prediction difficult or impossible.

While there are exceptions to the properties discussed in the following, they usually hold for real world data. If they do not hold it is no less complicated to handle the data.

1.2.1 Complex Structure

Real world data is complex.

In case of the telephone call example, one event can be expressed by four integers, one for the caller, the callee, the time, and the duration of the call. While this structure is still quite simple, modelling real events quickly increases the complexity.

When considering spatial objects the structure gets even more complex. We have streets, buildings, rivers, networks, ... and objects can be represented by points, lines, areas, polygons etc..

Simplified, the structure of a data object can be defined by a set of attributes and the structure may be classified by two criteria:

Dimensionality: the number of attributes of the objects

Common Structure: whether objects have a common structure or not

Usually the term *dimensionality* refers to the number of attributes uniquely denoting the remaining attributes. The dimensions form a multidimensional space referred to as *universe*.

Data objects sharing a common structure can be handled more easily, since the algorithms and management of such data can be specialized and tuned for its specific structure.

This is also the reason, why most computer programs require structured data as input in order to efficiently process it. Due to this, software developers were inventing file formats suiting their data/processing best. This lead to a mass of different formats and the problem of data exchange due to different formats. In the recent years, XML started to become popular as a commonly understood and accepted file format for storing semi-structured data, but its efficient management and indexing is still ongoing research [BRB03]. Curiously, promising techniques are restricted to documents sharing a common structure (a common DTD).

1.2.2 Skewed Data Distribution

Real world data is skewed.

When analyzing the distribution of sales, we observe that certain products are sold more often in certain shops and customers have preferences to buy specific goods. In rural areas there a fewer streets and buildings than in cities. Private phone calls occur more often on special holidays or during events.

In general, data is never distributed uniformly, sparsely populated universes with clusters of data are common. Additionally, logically independent dimensions might be correlated. The actual number of independent dimensions is usually significantly lower than the dimensionality of the universe. In literature this is often referred to as the *fractal dimensionality* and it can be a solution to solve the *curse of dimensionality* [PKF00], i.e., that the efficient querying of data becomes more complicated the higher the dimensionality becomes.

1.2.3 Huge Data Amounts

Real world data comes in huge amounts.

We have just started to collect data and every day the amount of new data increases. Considering phone calls in Germany, there are about 2 gigabyte of new data if we assume 80 million phones, 5 calls per phone and day and a storage requirement of 16 byte per call. Similar observations hold for sales. For moving objects like mobile phones or vehicles we get a mass of new data about their positions. In astronomy even huger amounts of data occur and researchers are not able to analyze data as fast as it is collected, so it is written to tapes and analyzed afterwards. Sensors and sensor networks open new sources for information and the expected data rates can be enormous.

1.2.4 Continuous Change

Real world data changes continuously.

While the past is static, things change over time and we want to record the change and learn from it. We are considering two classes here:

Snapshot Data: Only the currently valid data is interesting

Historical Data: We record all data. Old data is never changed[1] and new data is only added. We are interested in the evolution of things.

Depending on the application it might be sufficient to just have a snapshot, but often a historical view is of greater benefit. However, by considering just snapshots it is possible to tremendously reduce the amount of data that has to be managed and thus allows for managing and querying the data in an efficient way with an acceptable amount of required resources.

1.3 The Solution: Multidimensional Access Methods

The properties of real world data and the queries on it lead to a set of requirements that should be handled well by an indexing method that claims to efficiently support them. In the following we will use the term *access method* and *index* equivalently.

We have seen that data in general is multidimensional, it can have complex structure, comes in huge quantities with skewed distributions and it changes. Ad-hoc queries require reasonable guarantees for their execution time.

Database applications with a demand to handle this are e.g., data warehousing (DWH), online analytical processing (OLAP), geographical information systems (GIS), data mining, archiving systems, life-cycle management systems, etc.. Thus, an access method is needed which meets the requirements of all these applications and is also flexible and robust to support new applications.

Our focus is on multidimensional indexing in **r**elational **d**atabase **m**anagement **s**ystems (RDBMS). The UB-Tree [Bay96, Mar99] is an access method meeting the requirements closer than other access methods. It has proven its good properties in different application scenarios and has been fully integrated into a RDBMS kernel [RMF$^+$00].

1.4 Objective and Outline of this Thesis

The objective of this work is to show the superiority and flexibility of the UB-Tree in different applications. We complement the existing work in UB-Trees [Bay96, Mar99, Ram02, Pie03, Zir03] by a discussion on the advantages of the Z-curve used by the UB-Tree over other space filling curves and present algorithms for efficient incremental bulk updates. Further, we compare bitmap indexes with UB-Trees. Bitmap indexes also benefit from clustering data according to space filling curves, but we will show that the UB-Tree is still superior. We present a novel indexing technique developed by the author: The BUB-Tree, an enhanced version of the UB-Tree providing better performance for queries covering unpopulated space, which occurs in case of skewed data, i.e., most real world data sets. Finally, we will discuss techniques for the management of spatial objects with UB-Trees which are superior to existing spatial access methods.

[1]Data will not be changed except for rare exceptions, e.g., when correcting errors due to faulty transformations that occurred before loading the data.

Our theoretical considerations are complemented by a discussion of related work and performance measurements. Each chapter is closed by a summary of the results.

1.5 Related Work

In the following we survey related work in the field of multidimensional indexing and relational DBMS and query processing, which may be used for reference or further reading. Further, each chapter also includes a discussion of related work specific to the topics addressed in it.

1.5.1 RDBMSs

RDBMSs have been very successful in the last decades due to two key factors:

1. An generic and flexible model and a declarative query language, i.e., the relational algebra and SQL [Cod70].

2. Efficient support for the management of data and queries by robust and flexible access methods, e.g., B-Trees [BM72], B^+-Trees [Com79].

The access methods of a DBMS provide an efficient way to access a subset of the data selected by the restriction of a query. Online transaction processing (OLTP) applications have been the key business of RDBMSs in the past and their one-dimensional access pattern (e.g., point or range restrictions) are well supported by the B-Tree variants[BM72, Com79].

New applications (DW, GIS, etc..) have added a demand for efficient access to ranges of multidimensional point data. Adding multiple secondary indexes and performing an index intersection is one way to support them, but they impose additional maintenance cost. Additionally, they lack the ability to cluster multidimensional data and consequently the performance deteriorates dramatically with growing result set size due to a random access for each result row of the index intersection. Using a compound-B-Tree does not efficiently process multidimensional range queries either.

For point-restrictions it is also possible to utilize Hashing [FNP$^+$79] or Bitmap-Indexes [OQ97, CI98, WB98]. However, multidimensional range queries are not handled well by these due to the same reasons as for secondary indexes. In the case of Bitmap-Indexes the maintenance is even so costly that it is recommended to drop the indexes before performing updates and to recreate them afterwards [Ora00b].

1.5.2 Multidimensional Access Methods

Multidimensional access methods have been a research topic for the last couple of decades and research is still going on. [GG98] and [Sam90] give excellent surveys on almost all indexes known at their time. They differentiate between point access methods (PAMs) and spatial access methods (SAMs) for objects with an extent. Due to the huge amount of data, main memory access methods are not feasible.

The major focus of research was on spatial indexing and spatial joins especially for two dimensional spatial objects. R-Trees and their variants R*-Tree and R$^+$-Tree have been among the most accepted candidates as multidimensional successor of the B-Tree, however they lack performance guarantees for the basic operation. Due to overlap of the index nodes the R-Tree and R*-Tree cannot guarantee logarithmic cost for any of the basic operations. The R$^+$-Tree avoids the overlap by clipping leading to a higher cost during insertion and loosing storage utilization guarantees.

The Grid-file gives worst case guarantees, but has an extremely bad worst-case performance for update due to non-local splits. Furthermore it cannot handle dependencies in multidimensional data well, i.e., it does not guarantee a given storage utilization, which also holds for the k-d-B-Trees.

Approaches becoming more popular again in the last years are based on mapping the multidimensional space to one dimensional space. By reducing a multidimensional problem to a one dimensional, they allow for applying well known and tuned one dimensional access methods. Hence, insertion, deletion, point query and space utilization inherit the properties of the used one dimensional index, e.g., logarithmic guarantees for the basic operations and a storage utilization of at least 50% with B-Trees. Furthermore, techniques for concurrency and recovery are also inherited.

One way to map multidimensional points to one dimensional ones is based on space filling curves (SFC). A crucial property of the SFC is how well it preserves the spatial proximity of multidimensional points in one dimensional space. The choice of the SFCs depends on the application and for certain applications there are SFCs known to be best, i.e., for multidimensional spaces with equal domain in all dimensions the Hilbert-curve provides the best clustering.

Most approaches based on SFCs were designed for spatial data, e.g., the zk-d-B-Tree [OM84], XZ-Ordering [BKK99], DOT [FR91] and [FFS00]. [Law00] applies the Hilbert-curve for indexing of multidimensional data and provides algorithms for the basic operations.

1.6 The MISTRAL Project

This thesis is based on work done in the MISTRAL project (**M**ultidimensional **I**ndexes for **S**torage and for the **R**elational **A**lgebra) at the Bavarian Research Center for Knowledge-Based Systems (FORWISS).

The project investigates the improvement of the operations of the relational algebra with multidimensional index structures.

The major goal of the project was the implementation of the UB-Tree [Bay96, Mar99] and its applications. Numerous publications are available on the concepts of UB-Trees [Bay96, BM97, Bay97, Mar99], advanced query processing algorithms [Mar99, MZB99, FMB99, Ram02, Pie03, Zir03], applications [FKM$^+$00, Fen00, Pie98, RMF$^+$01, FMB02], and enhancements of the UB-Tree [Fen02, Wid04].

Cooperations with industry partners resulted in the integration of the UB-Tree into a DBMS kernel [RMF+00, TAS00] and actual applications with real world data. Cooperations with other academic institutes produced resulted for enhancement query processing algorithms [KTS+02, PER+03a, PER+03b].

The UB-Tree has proven to be a research effort producing papers, but it has also found actual applications, which is not true for many other multidimensional index structures.

The project homepage is at `http://mistral.in.tum.de` and provides further information, publications, downloads, and contact addresses.

This thesis ends the original MISTRAL project, which was dedicated to the research on multidimensional indexing and its applications by utilizing the UB-Tree. May the right index be with you ...

Chapter 2

Terminology and Basic Concepts

In this chapter the basic terminology used throughout this thesis is introduced . In most aspects it is conform with [Ram02] and [Mar99], but where appropriate we have modified it.

Definitions are provided for the terms used throughout the thesis. Topics occurring in more than one chapter are introduced here, e.g., query types, clustering, requirements for access methods.

The standard notation is used for floating point numbers. Quantifiers are $K = 10^3$, M 10^6, etc. A bit is denoted by Bit and a byte by B and quantifiers have the base 2 for bit/byte based numbers, e.g., $8Bit = 1B =$ one byte, $1024B = 2^{10}B = 1KB =$ one kilo byte, $MB =$ mega byte, $GB =$ giga byte and $TB =$ tera byte.

Time is measured in $sec =$ seconds, $ms =$ milliseconds, or $\mu s =$ microseconds.

2.1 General Notation

Our focus is on relational database management systems, but most of the results apply to database systems in general. We expect the reader to be familiar with the basic terminology of RDBMS:

Relation: Data is organized in a set of tables also called relations.

Tuple: Each row of a relation is considered as one event resp. unit of data called tuple.

Attributes: A relation has a fixed number of columns referred to as attributes and the value of an attribute have the same domain.

A tuple can be considered as a point in multidimensional space where the coordinates are given by the attribute values of the tuple. Consequently, we also refer to the attributes as dimensions. Often, only a subset of the attributes is used to qualify a tuple. Therefore, the attributes can be partitioned into two sets, a set of *qualifying attributes* or *indexing attributes* and a set of *informational attributes.*

In this thesis a tuple is considered to be a *point* in multidimensional space. The term *universe* is used to denote the multidimensional space defined by the Cartesian product of the domains of the attributes. Each attribute determines one *dimension.* The duality of tuple and point causes the following terms to be equal.

- domain of a relation, universe, multidimensional space
- relation, table, subset of a universe
- attribute, dimension
- attribute value, coordinate
- arity, dimensionality
- row, tuple, point

2.1.1 Tuple, Relation, Universe

Let $\mathbb{D}$ be a *domain*, i.e., a set of values, with a total ordering $<_{\mathbb{D}}$. $min_{\mathbb{D}}$ and $max_{\mathbb{D}}$ denote the minimal and maximal value of the domain. If the meaning is clear from the context we write $<$ instead of $<_{\mathbb{D}}$. $|\mathbb{D}|$ is the *cardinality* of $\mathbb{D}$.

Due to the limitation of hardware we have finite sets of values in real applications, i.e., most todays CPUs only support numeric data types with a size of 32 bits and memory and storage are limited.

Definition 2.1 (Direct Neighbor, $\doteqdot_{\mathbb{D}}$)
For any domain $\mathbb{D}$ with ordering $<_{\mathbb{D}}$ two values $a, b \in \mathbb{D}$ are neighbors, i.e., $a \doteqdot_{\mathbb{D}} b$, iff $(a <_{\mathbb{D}} b \wedge \nexists c \in \mathbb{D}$ with $a <_{\mathbb{D}} c <_{\mathbb{D}} b) \vee (b <_{\mathbb{D}} a \wedge \nexists c \in \mathbb{D}$ with $b <_{\mathbb{D}} c <_{\mathbb{D}} a)$. $\diamond$

Lemma 2.1 (Maximum number of neighbors)
A given value $a \in \mathbb{D}$ has at most 2 neighbors. $\diamond$

Proof 2.1 (Maximum number of neighbors)
The set of possible neighbors of a value a is $\{b | a \doteqdot_{\mathbb{D}} b\}$. There are at most two neighbors for $min_{\mathbb{D}} <_{\mathbb{D}} a <_{\mathbb{D}} max_{\mathbb{D}}$, otherwise there is only one neighbor, since $min_{\mathbb{D}}$ resp. $max_{\mathbb{D}}$ are the domain boundaries. $\diamond$

Definition 2.2 (Attribute, A)
An attribute A is a named domain, i.e., the name can be used to designated the domain. $\diamond$

Definition 2.3 (Tuple, $\vec{t}$)
A tuple $\vec{t}$ is a vector of n values $(t_1, \cdots, t_n)$ from the attributes $A_1, \cdots, A_n$ and all attributes have pairwise different names. $\diamond$

Definition 2.4 (Relation, R)
A relation R is a set of tuples with the same attributes. $\diamond$

Without loss of generality the attributes $A_1, \cdots, A_d$ with $1 \leq d \leq n$ are called *indexing attributes* and we also refer to them as dimensions $\mathbb{D}_1, \cdots, \mathbb{D}_d$. The remaining attributes $A_{d+1}, \cdots, A_n$ are called *informational attributes* and they are qualified by the indexing attributes. $|R|$ is the *cardinality* of R, i.e., the number of tuples in the relation R. For notational convenience we will refer to the set of possible dimension indices i as $D = \{1, \cdots, d\}$.

Definition 2.5 (Multidimensional Domain, Ω)
The *multidimensional domain* Ω of a relation R is the cross product of the d indexing attributes, i.e., $\Omega = A_1 \times \cdots \times A_d$. We say Ω has the dimensionality d. $\diamond$

Thus the *indexing attributes* of a tuple refer to a point in Ω. For tuples having only indexing attributes, we will use the terms *tuple* and *point* equivalently.

Definition 2.6 (Volume of Ω, $|\Omega|$)
The *volume* $|\Omega|$ of a universe is given by the product of the cardinalities of the domains, i.e., $|\Omega| = \prod_{i=1}^{d} |\mathbb{D}_i|$. $\diamond$

Definition 2.7 (Sparsity, $\xi(R)$)
Given a relation R of the multidimensional domain Ω, the *sparsity* of R is defined by $\xi(R) = 1 - \frac{|R|}{|\Omega|}$. $\diamond$

The sparsity of relational applications is typically greater than 99.9% [Ram02], thus most of the universe is not occupied by data points. The unoccupied empty space will be called *dead space*. There are also applications with a sparsity of 0 usually processing raster data, e.g., images, signals, etc..,

Definition 2.8 (Multidimensional Ordering, $<$)
For a given multidimensional domain Ω two tuples resp. points $\vec{p}, \vec{q} \in \Omega$ satisfy $\vec{p} < \vec{q}$, iff $\forall i \in D : \vec{p}_i <_{\mathbb{D}_i} \vec{q}_i$. $\diamond$

Two tuples are equal when all their attributes are pairwise equal. $\vec{p} \leq \vec{q}$ is used to denote $\vec{p} < \vec{q} \vee \vec{p} = \vec{q}$. While having a total order in one dimensional space there is no natural total ordering for multidimensional space that preserves neighbor relations as defined in the following.

Definition 2.9 (Multidimensional Direct Neighbor, $\doteqdot$)
For a given multidimensional domain Ω two points $\vec{p}, \vec{q} \in \Omega$ are *direct neighbors*, i.e., $\vec{p} \doteqdot \vec{q}$, iff $\exists i \in D$ such that $\forall j \in D \setminus \{i\} : p_i \doteqdot_{\mathbb{D}} q_i \wedge p_j = q_j$. $\diamond$

The number of direct neighbors grows linear to the number of dimensions.

Definition 2.10 (Multidimensional Neighbor)
For a given multidimensional domain Ω two points $\vec{p}, \vec{q} \in \Omega$ are *neighbors*, iff $\vec{p} \neq \vec{q}$ and $\forall i \in D : p_i = q_i \vee p_i \div_{\mathbb{D}_i} q_i$. $\diamond$

In contrast to *direct neighbors*, this also takes the neighbors at corners into account, e.g., more than one dimensions differs by one. There are $3^d - 1$ such neighbors, i.e., a d dimensional cube of side length 3 around $\vec{p}$ without the center $\vec{p}$. Consequently, the number of neighbors grows exponentially w.r. to the number of dimensions. This can be seen as the real reason for the curse of dimensionality.

Lemma 2.2 (Maximum and minimum number of neighbors of $\vec{p} \in \Omega$)
A given point $\vec{p} \in \Omega$ has at most $2d$ neighbors and at least d. $\diamond$

Proof 2.2 (Maximum number of direct neighbors of $\vec{p} \in \Omega$)
According to Definition 2.9 on the preceding page two neighbors are equal in all dimensions except one dimension i and there are d possible values for i. For a given i there are at most 2 neighbors with respect to $\mathbb{D}_i$ as we have seen before, since if a point is at the start or end of a dimension it has only one neighbor w.r. to this dimension otherwise it has two neighbors. $\diamond$

2.1.2 Query, Result Set, and Selectivity

In the following we define the terminology for queries on a relation. We limit our view to selection queries resulting in a subset of the relation, i.e., projection, sorting, and aggregation are not considered. Spatial queries are not considered here, since relations in their definition as given before, consist only of points. For further reading on the management of spatial data the reader is referred to Chapter 7.

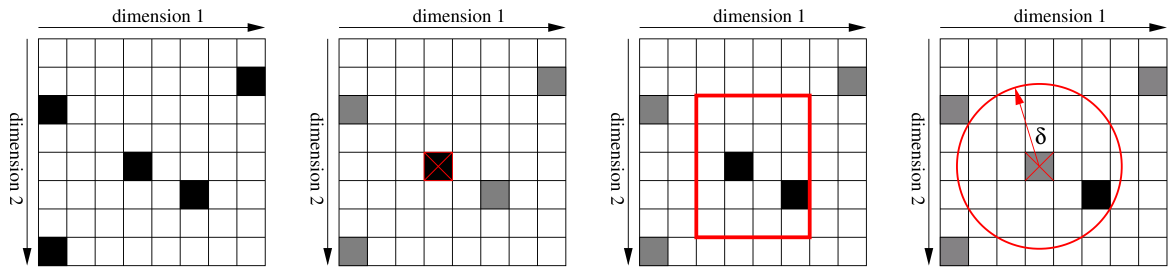
Definition 2.11 (Query, Q)
A *query* $Q \subseteq \Omega$ is a subset of the multidimensional domain defined by predicates. $\diamond$

Definition 2.12 (Result Set, $Q(R)$)
The *result set* $Q(R)$ is the subset of tuples of R within the query, i.e.,
$Q(R) = \{\vec{t} \in R | \vec{t} \in Q\} = R \cap Q$. $\diamond$

Definition 2.13 (Selectivity of a Query, $sel(Q)$)
The *selectivity* a of query is defined by the fraction of the result set size over the size of R, i.e., $sel(Q) = \frac{|Q(R)|}{|R|}$. $\diamond$

Definition 2.14 (Volume of a Query, $vol(Q)$)
The *volume* a of query is its cardinality, i.e., $vol(Q) = |Q|$. $\diamond$

Queries can be classified according to their predicate type, i.e., we have the following three basic multidimensional query types which are also depicted in Figure 2.1:



(a) Universe (b) Point Query (c) Range Query (d) NN Query

Figure 2.1: Example for a two dimensional Universe and Queries on it

Point Query: $Q_P(\vec{p}) = \{\vec{q} \in R | \vec{q} = \vec{p}\}$ for a given $\vec{p} \in \Omega$.

Range Query: $Q_R(\vec{l}, \vec{u}) = \{\vec{q} \in R | \vec{l} \leq \vec{q} \leq \vec{u}\}$. for a given $\vec{l}, \vec{u} \in \Omega$.

Nearest Neighbor Query: $Q_{NN}(\vec{p}, \Delta, \delta) = \{\vec{q} \in R | \Delta(\vec{p}, \vec{q}) \leq \delta\}$ for a given $\vec{p} \in \Omega$, distance function Δ and maximum distance δ.

A *range query* is specified by lower $\vec{l}$ and upper $\vec{u}$ bound with $\vec{l} \leq \vec{u}$, which restricts dimension i to the interval $[l_i, u_i]$. Therefore, a range query is a multidimensional interval $[\vec{l}, \vec{u}]$ which corresponds to an iso-oriented rectangular subspace of the universe.

A *point query* is a special case of a range query where lower and upper bound are equal, i.e., $\vec{l} = \vec{u}$. Queries restricting only a subset of all possible dimensions are called *partial match queries*.

By combining multiple range queries one can compose arbitrary query shapes [FMB99]. Other query types require additional predicates, i.e., the nearest neighbor queries require a distance function.

Example 2.1 (Relation, Universe, Queries)
The two dimensional universe in Figure 2.1(a) corresponds to the relation R with two integer attributes $A_1 = A_2 = [0, 7]$. The volume of Ω is $|\Omega| = |A_1| \cdot |A_2| = 8 \cdot 8 = 64$. The relation consists of the points $\{(0, 2)$, (7,1), (3,4), (5,5), $(0, 7)\}$ (black squares in Figure 2.1(a)) and thus its sparsity is $spar(R) = 1 - \frac{|R|}{|\Omega|} = 1 - \frac{5}{65} \approx 92\%$. Figure 2.1(b) depicts a point query $Q_P(3, 4)$, Figure 2.1(c) a range query $Q_R((2, 2), (5, 6))$ and Figure 2.1(d) a nearest neighbor query $Q_{NN}((3, 4), \Delta, \delta)$ with Δ being the Euclidean distance and δ the maximum distance. $\diamond$

2.2 Storage Devices

For the discussion of access methods it is also necessary to be aware of the different access types and physical organizations of data on different storage devices. Operating systems usually only provide interfaces for linear access to storage devices, although the devices

	RAM-Cache	RAM	Hard-Disk	Tape
average capacity	KB-1MB	MB-20GB	≈160GB	TB
random access	2ns	10-60ns	5-10ms	–
sequential access	2ns	2-10ns	1ms	min
EUR/GB[+]	-[*]	203.13	0.74	0.17

[+] Cheapest prices at http://www.alternate.de in July 2004.

[*] Memory caches are nowadays integrated on the CPU-chip, thus we cannot provide a price here.

Table 2.1: Properties of different Storage Classes

might not be organized in a linear fashion. This is done for convenience and allows programs to be independent from the structure of the underlying device.

Based on the relation of the positions of successive accesses, two access patterns are distinguished:

Random Access: The position of successive accesses is independent from each other.

Sequential Access: The position of successive accesses is linear dependent, i.e., the next access continues at the position on the disk where the last access stopped.

We have a variety of different storage devices used in todays computers. The fastest storage is tightly coupled with the CPU, but it has only a very limited capacity and is very expensive. With growing capacity access usually slows down and the cost per byte drops. Three major classes are commonly used and their properties with respect to capacity and access times are listed in Table 2.1:

Primary Storage: Registers, Caches, RAM

Secondary Storage: Hard disks, CD-Roms, DVDs, USB-Sticks, and Floppies

Tertiary Storage: Tapes (with tape-robots)

In general there is no random access device providing the same access times for random and sequential access. However, if random access is not substantially slower than sequential devices are called *random access devices.*

Example 2.2 (Main memory is not literally RAM)

CPU caches and main memory are regarded as random access devices. Standard RAM has an access time of 60ns (nano seconds) for random access and 10ns for sequential accesses. Usually, CPU and memory chips are communicating via a special system bus and communication has to be synchronized by handshakes. With random access each address is transferred to the memory chip and then the CPU waits for the data to be available on the bus. Sequential access is supported by a burst mode of the memory chips, i.e., the CPU makes only a single request with a starting address and the size of the requested data block. No further address transfers (communication and sync) are necessary and while transferring data, the memory chip already fetches subsequent data items. $\diamond$

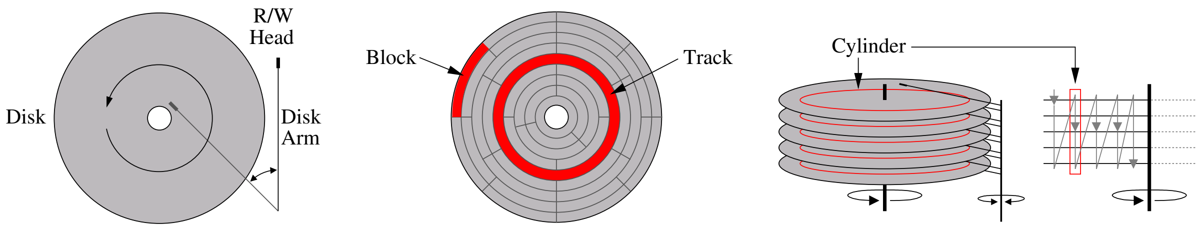
Main memory is regarded as a random access devices. Also disks provide random access, but it is substantially slower than sequential access. Tapes are typical sequential access devices where positioning the read/write head for an random access may take several minutes. Larger storage solutions consist of robots changing tapes and thus take even longer for random access. Lately a new and much cheaper and faster solution for huge amounts of data has been developed by utilizing clusters of cheap personal computers with hard-disks. This system is running as a backup solution at the university of Tübingen and there is an interesting article on *slash.dot* about this [Gui03].

Our focus is on access methods for secondary storage, where random access is substantially more expensive than sequential access. The design of an efficient access method has to take into account these properties of a storage device. This thesis focuses on disk-based accesses methods, so in the following disk-based storage is discussed.

Figure 2.2(a) depicts the typical architecture of a hard-disk drive. Information is stored on a disk coated with a magnetic surface, which can be modified by a read/write head (R/W-head). The disk is spinning and its surfaces are partitioned into tracks and the tracks consist of a group of blocks also referred to as sectors. A block is the smallest addressable and transferable unit of a disk. Block sizes vary between 512 byte and a few KBs. To gain a higher capacity, todays hard disks have more blocks on the outer tracks than on the inner tracks.

In order to access a block, the disk arm positions the R/W-head on the correct track and then waits for the block beginning to appear below it while the disk rotates (latency). The time required for this operation is called *positioning time* t_π. The block is now read and transferred into main-memory, requiring a *transfer time* t_τ. Blocks and even whole tracks are usually cached within the drive and also operating systems usually save some of the following blocks in their caches. This technique is called *prefetching*. Once the R/W-head is positioned on a track, subsequent block accesses require only t_τ, since no further positioning is necessary. Therefore, the total time to read one block by random access is $t_\pi + t_\tau$, but for a sequential access it is just t_τ. Hard disk vendors provide information about three different positioning times, i.e., the average, a single track movement and full stroke from the first to the last track.

Hard disks are organized as stacks with a R/W-head on all surfaces of the disks (Figure 2.2(c)). The disks and R/W-heads are fixed to each other and thus are moving only jointly. This allows for parallel reading from all disks of the stack and increases the transfer rate and the capacity per logical track (cylinder), but the time for positioning remains the same. In the last decade, hard-disk producers enabled increasing capacities by narrower tracks and shorter blocks due to the size-reduction of the R/W-head. This has increased the capacity and the transfer rate to a large extent while the positioning time has been reduced only slightly.



(a) Physical Elements (b) Partitioning into Tracks/Blocks (c) Disk Stack

Figure 2.2: Physical Structure of a Hard-Disk

Consecutive blocks are grouped into clusters by operating systems or pages by DBMSs which are used as larger units of disk access. Therefore, the cost for accessing a disk is usually not measured in the number of block accesses, but in the number of page accesses.

Definition 2.15 (Page)
A *page* P is a container of fixed size. ◇

All pages of a database have the same size. A page is usually allocated as a set of consecutive disk blocks. Besides tuples, a page also stores additional management information, e.g., its page number, references to other pages (e.g., next, previous, father, child, brother), its type (e.g., data, index, etc.), informations on its content (e.g., number of tuples, type of tuples, free space, occupied space, etc.).

Definition 2.16 (Page capacity and page utilization)
With $P.size$ we denote the space available for storing tuples. With a fixed size $tsize$ required to store an actual tuple on a page and $P.n$ being the number of tuples stored on a page, the page capacity $P.C$ and the page utilization $P.U$ are defined as follows:
$P.C(tsize) = \left\lfloor \frac{P.size}{tsize} \right\rfloor$ and $P.U = \frac{P.n}{P.C(tsize)}$ ◇

The cost for a sequential page access is $t_\sigma(n) = n \cdot t_\tau$ and for a random page access it is $t_\rho = t_\pi + n \cdot t_\tau$ where n is the number of blocks a page consists of. Todays hard drives have an average positioning time between 3.4ms - 8ms and a transfer rate between 36.6MB/sec - 52.8MB/sec [IBM01, Sea01]. With growing page size the percentage of the positioning cost t_π of the total cost t_ρ decreases (Figure 2.3), but with typical page sizes between 2KB and 1MB it is still a significant part of the access time. In order to optimize the access to a device we have to minimize the number of accesses in general, but in particular random accesses should be transformed into sequential ones.

Figure 2.4 on page 20 shows the access times on different hardware and operating systems for accesses to 2KB pages. We have measured them for two configurations, a notebook with Windows 2000 and a Fujitsu disk, and a Sun Sparc 5 with Solaris 2.8 and a Seagate. The internal disk caches were not disabled, but the Seagate disk was mounted in direct I/O mode and thus the caching of the operating system was disabled. As expected,

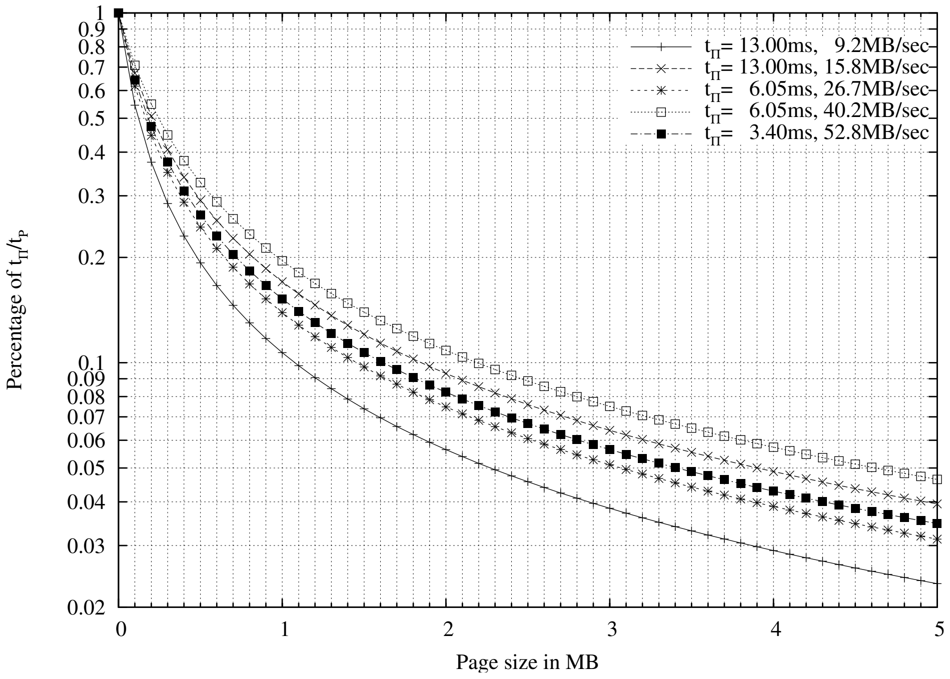


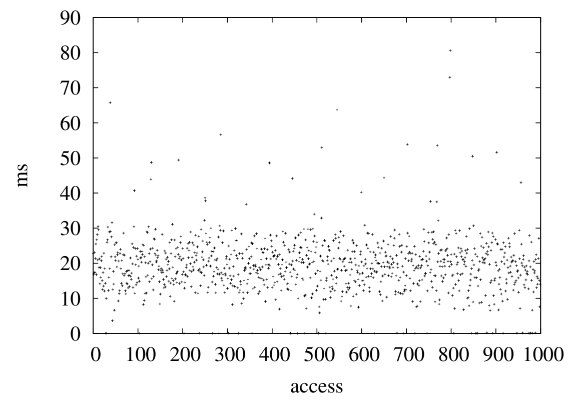
Figure 2.3: Percentage of Positioning Cost with growing Page Size

sequential accesses are faster than random accesses by orders of magnitude (Table 2.2). Additionally, all sequential accesses require nearly the same time. There are a few outliers due to other operating system accesses to the disks, which could not be eliminated.

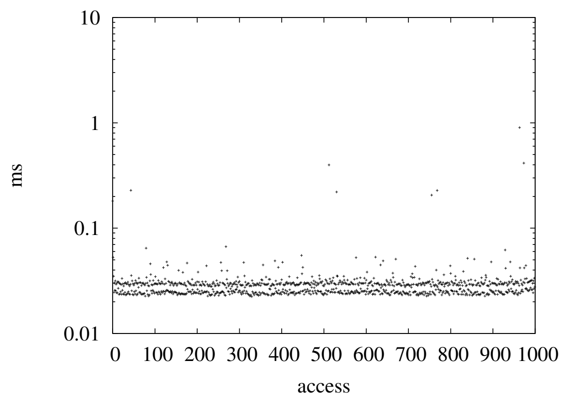
Figure 2.4(c), and 2.4(d) reflect the characteristics [Sea01] of the Seagate drive, i.e., its faster random access time (6.05ms vs. 13ms for the Fujitsu drive shown in Figure 2.4(a),2.4(b)) and its lower latency ($\approx$2.99ms vs. $\approx$7.14ms). The Seagate disk under Sun/Solaris has two main clusters (0.01ms and 0.37ms), but 85% of all accesses take only 0.01ms which are probably blocks on the same disk and the others when switching to the next disk. There are also peaks where the R/W-head has to be moved, i.e., when moving it to the next track. Every $\approx$ 2000 accesses there is a peak, i.e., after reading $2000 \cdot 2\text{KB} \approx 4\text{MB}$ which fits to the properties of the disk with an average capacity of 215KB per track and 24 heads we get $215\text{KB} \cdot 24 \approx 5.1\text{MB}$ including administrative file system data. Furthermore, the actual track capacity is not uniform, but it was less than the average track capacity, since the data was located on inner tracks. The sequential reads on the Seagate disk under Solaris are slower in average than for the Fujitsu disk as it was mounted in direct I/O mode, thus preventing pre-fetching by Solaris.

system	access type	# of pages	# of accesses	min	max	average
Notebook + Win2k	Random	51200	1000	0.050	80.61	19.33
Notebook + Win2k	Sequential	51200	1000	0.022	0.90	0.03
Sun Sparc + Solaris	Random	51200	5000	0.420	19.51	5.43
Sun Sparc + Solaris	Sequential	51200	5000	0.009	17.64	0.11

Table 2.2: Minimal, maximal and average Access Times in ms for 2KB Pages

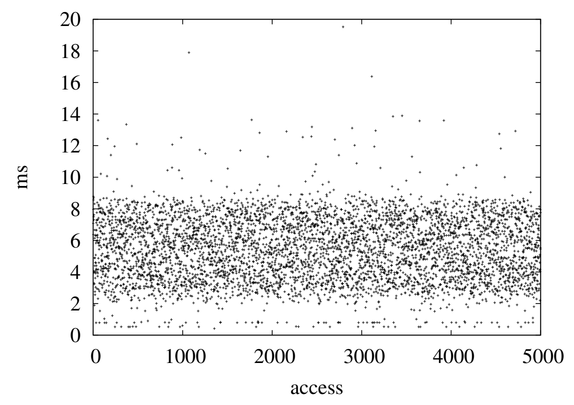


(a) random

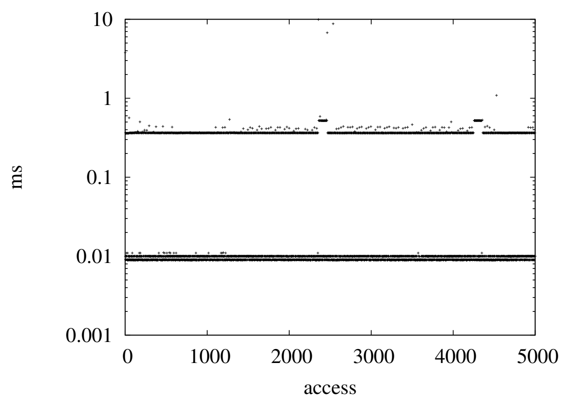


(b) sequential

Dell Latitude CPt, Celeron 466MHz, Win2k, Fujitsu MHH2064AT:
6.4GB; 2 disks; 4 heads; track capacity 107,008 to 185,856 bytes; sector size 512 bytes; cylinders 11,172; sectors/track 209 - 363; seek: track2track 1.5ms, average 13ms, max 23ms; latency ≈7.14ms; transfer rate 9.2 to 15.8 MB/s for sequential access



(c) random



(d) sequential

Sun Sparc 5, Solaris 2.8, Seagate ST173404LW/LC:
73.4GB; 12 disks; 24 heads; track capacity ≈ 215,248 bytes; sector size 512 bytes; cylinders 14,100; sectors/track ≈420; seek track2track 0.8ms, average 6.05ms, max 15; latency ≈2.99ms; transfer rate 26.7 to 40.2 MB/s for sequential access

Figure 2.4: Random and sequential Access Times on different Hardware Configurations

2.3 Caching

Caching is used to minimize the access to the underlying storage device. It optimizes sequential and multiple accesses to data. In case of storage devices this also minimizes the effects of random accesses, i.e., if a pages is in the cache it is not necessary to fetch it from the underlying storage again. There is a cache for each device in the storage hierarchy.

Hard disks have internal caches, to speed up sequential access to all blocks of a track. When positioning the R/W-head on a new track its content is read into cache by one rotation. Any block on that track can now be served from the cache without causing further latency due to waiting of the R/W-head reaching this block.

Between CPU and RAM there are usually several caches. They are faster, more expensive, and smaller the nearer they are to the CPU. The last cache levels are often located on the CPU-chip itself. Code with a high locality (accessing only a small range of addresses) causes most of the time only cache accesses. Jump prediction is used to fetch needed data into the cache in advance.

Additionally, operating systems (OSs) perform their own caching taking specific knowledge into account. OSs know the location of file fragments on disk and thus can read the fragments in advance to speed up sequential accesses. This technique is known as *prefetching.*

Example 2.3 (Linux Disk Cache)

Linux utilizes all available free physical memory to cache disk accesses. With sufficient RAM, multiple accesses to files will cause a disk access only once, which improves repeated program compilation and document generation with LaTeX as those are reading the same include files again and again. ◇

Database management systems and other applications, are able to take specific application knowledge about access patterns into account, e.g., during range query processing utilizing an index, the index pages are kept in cache as they are accessed multiple times.

As caching may prevent the bad effects of random accesses it is important to take the actual size of a database into account. Random accesses on databases smaller than the available cache will cause higher accesses times only until completely cached, but not for subsequent accesses. Only databases significantly larger than the caches will exhibit the effects of random accesses to secondary storage.

2.4 Clustering

The limiting factor for the performance of applications processing non sequential accesses to databases not fitting into main memory is I/O. Those applications spend most of their time waiting for secondary or tertiary storage I/Os to be finished.

Clustering in conjunction with indexing is the key to significantly reduce the I/O cost by reducing the number of random disk accesses and disk accesses in general. The objective of clustering is to store data, that is likely processed together, physically close to each other, e.g., on the same page, and thus replace a set of random I/Os by fewer sequential I/Os. Thus a query defines what tuples are processed together and its access order to the tuples defines an ordering of the tuples. Usually there are classes of queries defining the same order.

It is impossible to find a clustering suiting all possible queries and therefore one has to make a decision for a specific clustering or adapt the clustering during query processing. Since it is tedious to adapt with every new query, most systems require a decision for one clustering order or a reorganization once in a while. We distinguish three types of clustering:

Definition 2.17 (Tuple Clustering)
Tuple clustering stores tuples on pages in the order defined by the clustering. $\diamond$

Tuple clustering does not force the tuples to be stored in clustering order on the page. However please note, with a page containing tuples between $[s : e]$ w.r. to clustering order, there is no other page containing tuples within $[s : e]$.

Definition 2.18 (On-Page-Tuple Clustering)
On-Page-Tuple clustering stores tuples by *tuple clustering* and additionally also the tuples on the page in clustering order. $\diamond$

Definition 2.19 (Page Clustering)
Page clustering stores pages in the order defined by the clustering. $\diamond$

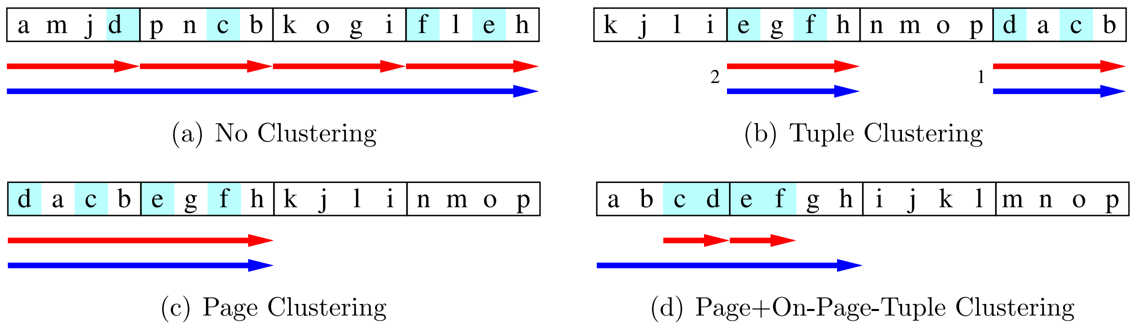
The different types of clustering have distinct properties with respect to maintenance and query performance. As maintenance operations we are considering insertion and deletion. As queries we consider those queries defining the clustering order.

In case of no clustering we have to read all pages and filter their content. Pages are read by a sequential scan benefitting from the prefetching/caching of disks and OS, but inspecting every tuple causes a high CPU cost. Inserting new tuples and pages is simply handled by appending to the end. If we do not access the data anymore, this is definitely the best solution. Deletion and update require a complete scan of all pages to locate the affected tuples.

In the following, we assume there is some kind of index identifying the pages we have to inspect in order to manage and query data.

In case of *tuple clustering*, random page accesses for each tuple are avoided, but within a page each tuple has to be inspected causing additional CPU cost. Inserts can be efficiently performed by appending to a page.

On-page-tuple clustering increases the cost during insertion and update due to maintenance of the order, but reduces the CPU-cost during query processing as binary search can be used to quickly identify queried ranges. With increasing page size this can have a significant effect on the query performance.


a m j d | p n c b | k o g i | f l e h
k j l i | e g f h | n m o p | d a c b
2
1
d a c b | e g f h | k j l i | n m o p
a b c d | e f g h | i j k l | m n o p

(a) No Clustering (b) Tuple Clustering (c) Page Clustering (d) Page+On-Page-Tuple Clustering

A page has a capacity of 4 items. The first line of arrows indicate tuple ranges on a page, the second arrow ranges of pages, that are accessed sequentially.

Figure 2.5: Different Types of Clustering for Lexical Order and the Interval Query $[c, f]$

With *page clustering*, adding new pages requires to move pages in order to get a gap, unless when adding to the end. In worst case, when inserting a new page at the head, every page has to be read and written once.

With *on-page-tuple* and *page clustering* at the same time the best possible query performance is reached, since there is only one interval of pages and tuples. However, it is causing the highest maintenance cost and thus is only suitable for applications having a "write once and read many times" access pattern.

When using files to store a database it is not guaranteed that page clustering actually physically clusters data on the disk as desired, since depending on the file-system a file can be stored in different fragments. This can be fixed by defragmentation tools, but the location of new pages is not predictable. Therefore, DBMSs also offer to access disks/partitions directly and thus guarantee the location of pages.

Example 2.4 (Tuple Clustering, Page Clustering)

Figure 2.5 depicts different types of clustering for the tuple set of characters $\{a, \cdots, p\}$ and the lexical order, i.e., for the class of queries restricting the data to an interval. We see pages with a capacity of 4 and a page utilization of 100% ordered from left to right as stored on disk. Below the pages (rectangles with four items in them) the set of ranges is indicated with respect to tuples and pages that have to be accessed. Each arrow indicates a sequential access on its cover.

Figure 2.5(a) depicts no clustering, since we do not have any information where the relevant tuples are stored, we have to access all pages by a sequential scan and filter their content for result tuples. Figure 2.5(b) depicts tuple clustering, i.e., we have to access two pages requiring two random accesses and have to scan the whole page for result tuples. Figure 2.5(c) depicts page clustering, i.e., there is only one range of pages that can be read sequentially causing one random I/O, however the page content has to be post-filtered. Figure 2.5(d) depicts page and on-page-tuple clustering at the same time, allowing for reading one consecutive set of pages and tuples and thus causing the least overall costs. ◇

2.5 Access Methods

As said before, in order to utilize clustering an index, also referred to as access method, is necessary. An index provides a way to organize and query data, i.e., it helps to efficiently find queried data.

Different storage devices also require different access methods in order to be accessed efficiently. In general, primary storage data structures try to minimize the number of tuples to process, while secondary storage index structures are minimizing the number of accesses to secondary storage. For example, AVL-Trees perform well for main memory since they are balanced binary trees, but B-Trees are superior for secondary storage due to their higher degree of branches per node and thus they are substantially smaller with respect to height and reduce the number of I/Os. This is crucial since accessing the secondary storage is the limiting factor and usually not the processing of tuples.

2.5.1 Clustering Index, Primary Index, Secondary Index

Without an index, data can be only accessed unqualified, i.e., in order to find relevant tuples the whole table has to be scanned. In the relational world this is called *full table scan* (FTS) or short *scan*.

A *clustering index* defines the physical organization of data , i.e., its clustering, and provides direct access to it. There can only be one clustering index for one instance of data and usually it is equivalent to the *primary index*, i.e., if they have the same key. Any other index on the same instance of data is a *secondary index* and has no effect on the clustering.

Secondary indexes require an additional indirection, i.e., they return tuple references rather than tuples. Possible tuple references are:

- keys of another index, usually the primary index (tuple id: TID)
- references to pages containing the relevant tuples
- references to pages plus the positions of tuples on the pages (row id: ROWID)

Using references to pages allows to circumvent the primary index during tuple fetching, but it requires additional efforts to correct the references when tuples are relocated due to page splits or merges.

Figure 2.6 shows a primary index referring to pages and a secondary index referring to the tuples stored on the data pages of the primary index. Data pages are displayed in physical order and we assume the secondary index has a different order than the primary index. It is obvious that a range query on the secondary index will cause random accesses when fetching the tuples.

One way to circumvent the random accesses is to include all queried attributes in the secondary index, i.e., make it a *covering index* for the queries that utilize it. Another

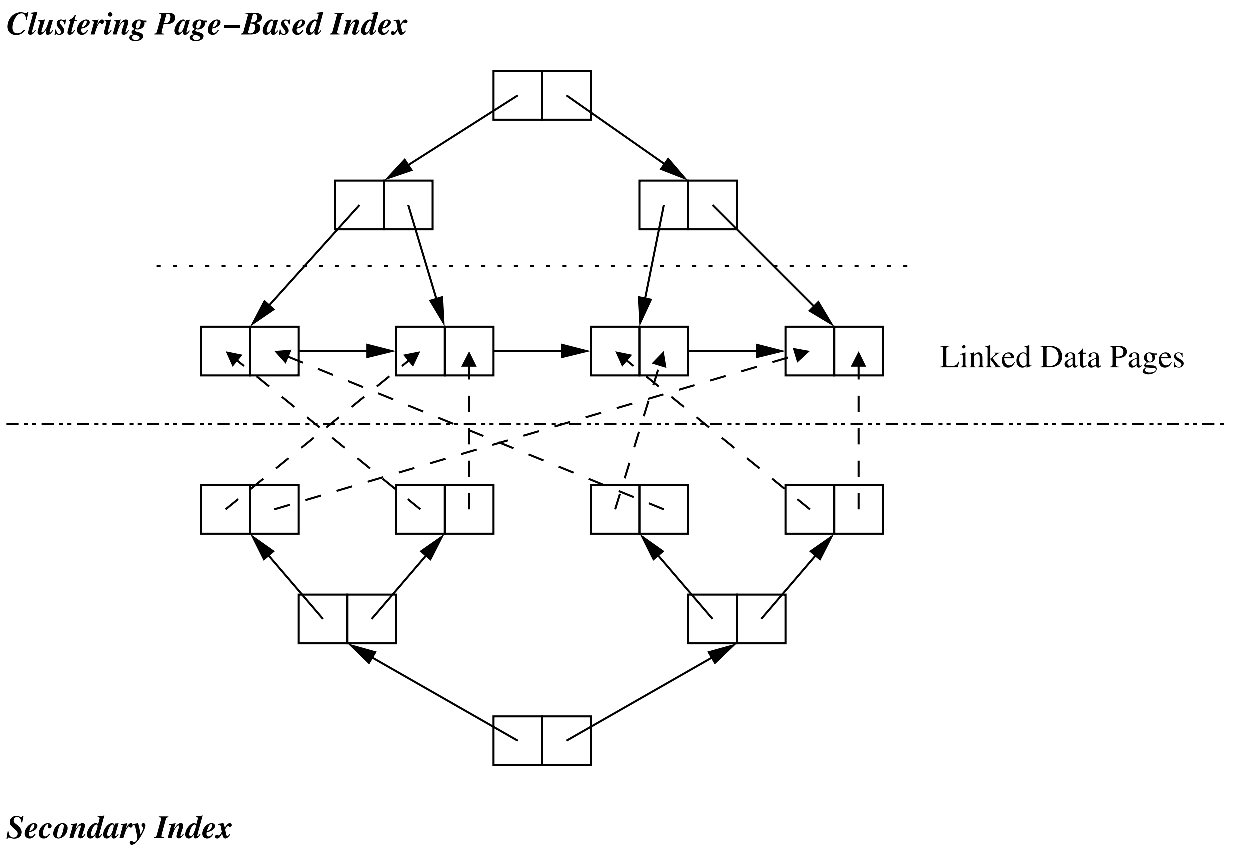


Figure 2.6: *Clustering Page-Based Index* vs. *Secondary Index*

option is to postpone the access until all (or a substantial amount) of the TIDs/ROWIDs are available and to sort them ascending before actually fetching the tuples.

Further, a clustering index allows to index pages instead of tuples. This allows for a smaller index and better performance due to the page based processing. Pages lying within a queried range can be returned for further processing without inspecting each tuple. Secondary indexes do not allow this optimization since they are not clustering.

2.5.2 Address

To distinguish between the key of a relation and the key of an index we will refer to the latter one as address. Without loss of generality we define an address used by an index as follows:

Definition 2.20 (Address $\alpha(\vec{p})$ of a tuple $\vec{p}$)

Let $\mathbb{A}$ be the domain of addresses. The address function $A : \Omega \rightarrow \mathbb{A}$ maps each tuple $\vec{p} \in \Omega$ of a relation R to an address $\alpha \in \mathbb{A}$. $\diamond$

Note that addresses may not be unique and thus an address function is not necessarily a bijective function. In the following addresses are denoted by Greek letters.

An example for an address is a B-Tree key, which is a single attribute of a tuple or the concatenation of multiple attributes, usually referred to as *compound* or *composite* key. Another example are hashing functions mapping tuples to buckets, thus the address is the bucket id.

Internally indexes usually use a different representation of *addresses*. A universal internal representation of addresses is a bit-string. An index supporting this representation requires only the definition of transformations from the attribute type to its bit-string representation and thus does not need to know anything about the actual data type.

Furthermore, bit-strings have no size limit like fixed size types (e.g., integer, float, etc.) and bit-operations are supported efficiently by assembler code. A bit-string corresponds to a positive integer number and thus we may use this representation where appropriate. $|\alpha|$ denotes the length of an address in bits.

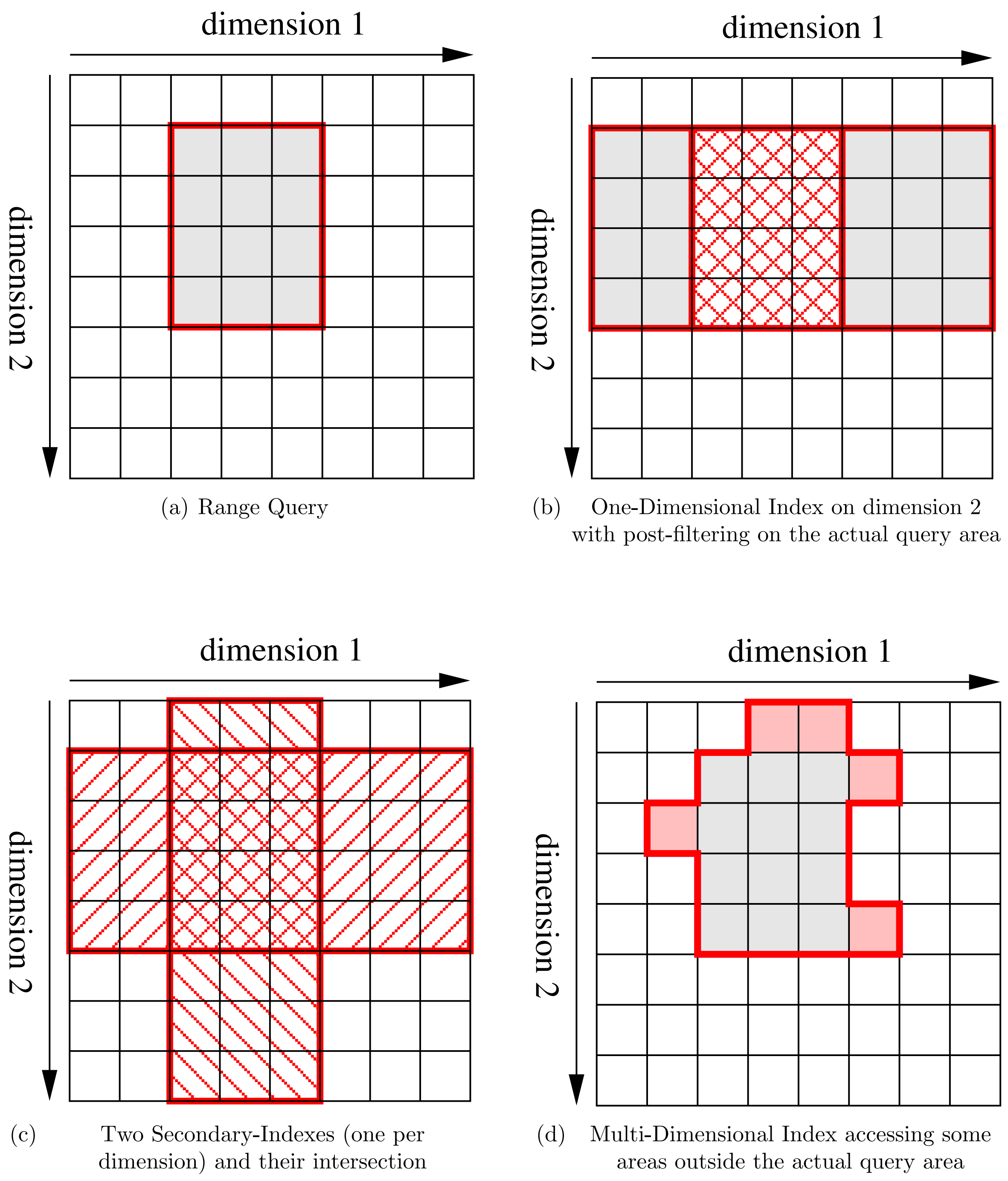
2.5.3 Accessed Data

During query processing the actually accessed data differs depending on the used indexing technique which are depicted in Figure 2.7. A perfect index would access only the queried data (Figure 2.7(a)), but this does not happen in general.

A one dimensional index is only able to utilize the restriction on one dimension and has to apply post-filtering to the result as depicted in Figure 2.7(b). In worst case it has to read the whole table, i.e., if a partial match query does not restrict the index dimension.

With secondary indexes there is one index for each dimension. Query processing is performed by utilizing the restriction on each index separately and then the results are intersected. The results are usually tuple identifiers. Finally, the result tuples are fetched by random point queries (middle part of Figure 2.7(c)). Sorting the tuple identifiers according to the primary index before accessing the data can speedup the access by reducing the randomness, but it breaks pipelining, as fetching the tuples can start only after sorting.

A multidimensional index utilizes the restrictions in all dimensions and in average loads some “small” fraction of additional data besides the actual query shape (Figure 2.7(d)).



(a) Range Query

(b) One-Dimensional Index on dimension 2 with post-filtering on the actual query area

(c) Two Secondary-Indexes (one per dimension) and their intersection

(d) Multi-Dimensional Index accessing some areas outside the actual query area

Figure 2.7: Range Query and actually affected Area of the Universe

Chapter 3

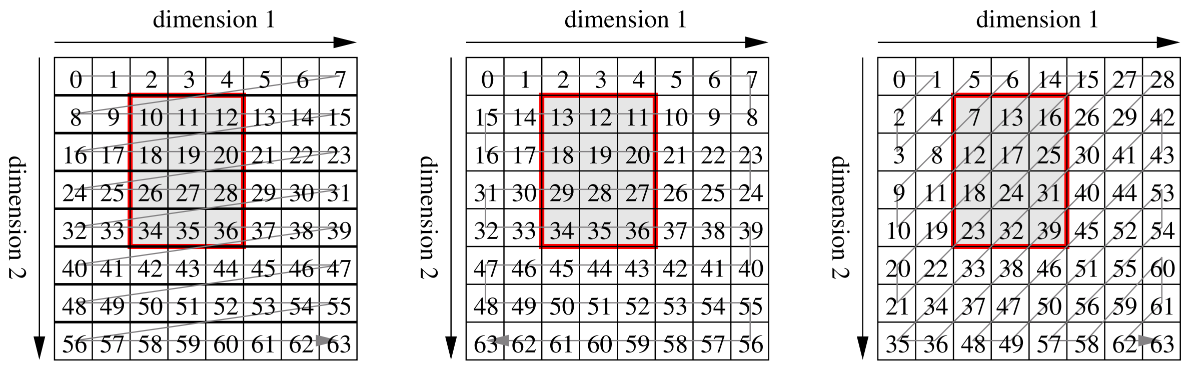
Space-filling Curves

A space-filling curve (SFC) is a function mapping the multidimensional space into the one-dimensional space. It passes through every cell in multidimensional space so that each cell is only visited once. Neighboring points on the SFC are not necessarily neighbors in multidimensional space, as a SFC may not be continuous, but it can have jumps. Thus, a SFC is a way to calculate an address for a tuple.

Definition 3.1 (Space Filling Function)
We call a function $S : \Omega \to \mathbb{S} \subseteq \mathbb{N}_0$ a space filling function, iff $S(\Omega) = \mathbb{S}$ is a bijective function projecting the values in Ω to $[0, \cdots, |\Omega| - 1] = [0, \cdots, |\Omega|[$. $\diamond$

We use $\alpha = S(\vec{p})$ to denote the SFC-address α of a tuple $\vec{p}$ and S^{-1} to denote the inverse function, i.e., the function calculating a point from an SFC-address. Due to S being bijective $\vec{p} = S^{-1}(S(\vec{p}))$ holds.

Connecting the points in Ω that correspond to adjacent values in $S(\Omega)$ draws the curve. Figure 3.1 shows three example curves which will be discussed in later sections.



(a) Compound-curve (b) Snake-curve (c) Zig-Zag Curve

Figure 3.1: Simple SFCs for 8x8 universe with range query

SFCs have been discovered by Peano [Pea90] and thus are often referred to as Peano-curves. Hilbert [Hil91] generalized the idea to a mapping of the whole space. An introduction and comparison on the most popular SFCs follows. For a historical survey and other SFCs the reader is referred to [Sag94].

A SFC provides a linear order for a discrete finite multidimensional space and therefore imposes a specific one-dimensional clustering of data. Depending on the application, one can choose the SFC providing the best clustering with respect to the application needs. Most multidimensional applications benefit if the spatial proximity of points is preserved as well as possible, i.e., points which are near to each other in multidimensional space should also be near to each other on the SFCs. Especially, for multidimensional range restrictions on Ω it is desired to cluster data in this way, since those points are likely to be accessed together.

In the following we define some metrics useful for defining properties of SFCs and to compare and evaluate them:

Definition 3.2 (Euclidean Distance, Δ)
The distance of two points in Ω, the Euclidean distance, is defined as
$\Delta(\vec{p}, \vec{q}) = \sqrt{\sum_{i=1}^{d} (p_i - q_i)^2}$ ◇

Definition 3.3 (Manhattan Distance, Δ_M)
The Manhattan distance of two points in Ω is defined as
$\Delta_M(\vec{p}, \vec{q}) = \sum_{i=1}^{d} |p_i - q_i|$ ◇

Definition 3.4 (SFC Distance, Δ_S)
The distance of two points on the space filling curve is defined as
$\Delta_S(\vec{p}, \vec{q}) = |S(\vec{p}) - S(\vec{q})|$. ◇

Spatial proximity can only be preserved to some extent, since mapping a higher dimensional space to a one dimensional space reduces the number of possible direct neighbors to two. Thus a SFC can preserve only two direct neighbors in best case, i.e., only for two of the $2d$ neighbors n_i, n_j of a given point p with distance $\delta = 1$ we have:

$$\Delta(\vec{n}_i, \vec{p}) = \Delta(\vec{n}_j, \vec{p}) = \Delta_S(\vec{n}_i, \vec{p}) = \Delta_S(\vec{n}_j, \vec{p}) = \delta \tag{3.1}$$

$$\text{with } i, j \in D \wedge i \neq j \text{ and } n_i \neq p \neq n_j \tag{3.2}$$

All other direct neighbors have a SFC-distance which is greater than 1. Consequently two neighbors on the SFC are not necessarily neighbors in Ω and in worst case their distance might be the maximum distance. This also holds for point with a distance $\delta > 1$ to a given point $\vec{p}$. Further, also the opposite might happen, i.e., the distance of two points might be δ while their SFC-distance is the maximum distance.

Figure 3.2 depicts three examples for possible relations of Δ_M to Δ_S for the two shaded points. Figure 3.2(a) depicts the ideal case, i.e., the SFC-distance is equal to the real

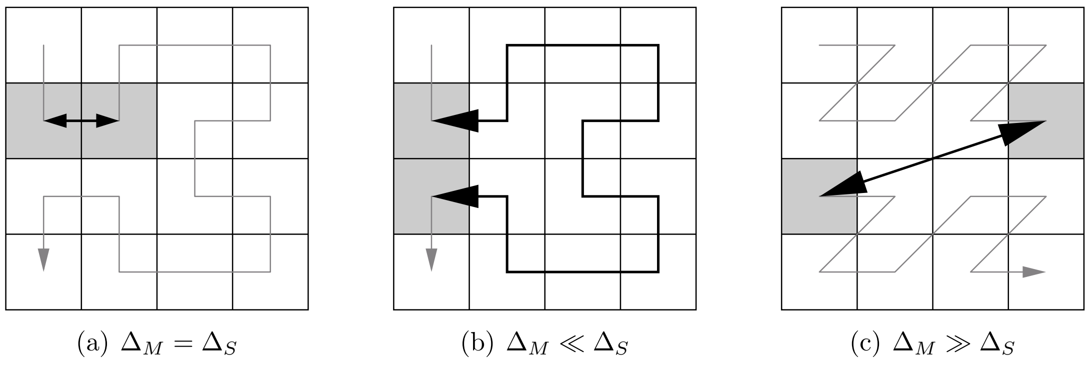

(a) $\Delta_M = \Delta_S$ (b) $\Delta_M \ll \Delta_S$ (c) $\Delta_M \gg \Delta_S$

Figure 3.2: Example relations of Δ_M to Δ_S

distance. In Figure 3.2(b) the actual distance is $\Delta_M = 1$ while the SFC-distance is $\Delta_S = 13$. In 3.2(c) the actual distance is $\Delta_M = 4$ while the SFC-distance is $\Delta_S = 1$.

Another important property of a SFC is whether it is order preserving or not. With respect to query processing it is an advantage if the SFC is order preserving since this can avoid or reduce the cost of an additional sorting step. Formally we define an SFC to be order preserving if two tuples are smaller w.r. to their address as follows:

Definition 3.5 (Order Preserving SFC)
An SFC is *order preserving* iff: $\forall \vec{p}, \vec{q} \in \Omega : \vec{p} < \vec{q} \Rightarrow S(\vec{p}) < S(\vec{q})$ ◇

Definition 3.6 (Dimension Order Preserving SFC)
An SFC is *dimension order preserving* iff:
$\forall \vec{p}, \vec{q} \in \Omega, i \in D, \forall j \in D \setminus i : p_i < q_i \wedge p_j = q_j \Rightarrow SFC(\vec{p}) < SFC(\vec{q})$ ◇

Thus if a point is greater than another point w.r. to a specific dimension then also its SFC address is greater.

For segments on a SFC we can define the following additional properties:

Definition 3.7 (Volume of a SFC-segment)
The *volume* of an interval $[\alpha, \beta]$ on a SFC is defined as $vol([\alpha, \beta]) = \beta - \alpha$ ◇

Definition 3.8 (Region of a SFC-segment)
The *area* in Ω covered by an interval $[\alpha, \beta]$ on a SFC is defined as
$R([\alpha, \beta]) = \{\vec{p} \in \Omega | \alpha \leq S(\vec{p}) \leq \beta\}$ which is called *region* of $[\alpha, \beta]$. ◇

3.1 Space Filling Curves

In this section we introduce commonly known curves. There are two important classes of SFCs with distinct properties:

Fractal/Recursive Curves: These are self-similar curves recursively subdividing Ω into smaller sub-spaces and applying a leitmotiv to the subspaces.

Curves constructed by Bit-Permutations: The one-dimensional value of a tuple is calculated by a permutation of the attribute values bit-representation.

When using the term *attribute values* in the context of SFCs we are considering normalized values, i.e., not p_i, but $N(p_i)$ where N is defined as bijective and order preserving mapping $N : \mathbb{D} \rightarrow [0, |\mathbb{D}_i| - 1]$. Thus the bit-representation of p_i is the bit-representation corresponding to the integer $N(p_i)$.

Fractal curves may also be applied to infinite domains and they allow to partition the universe at different granularities, i.e., at different levels of recursion.

The main advantage of bit-permutations is that they allow for a very efficient calculation of addresses of points and address calculation in general. Some of them also create fractal curves, e.g., bit-interleaving creates the Z-curve.

In the following we introduce common SFCs, we will discuss how to utilize a SFC for indexing, then we review related work and finally discuss and compare SFC properties w.r. to clustering and query processing for our application scenario, i.e., sparsely populated multi-dimensional universes.

3.1.1 Compound Curve

The compound curve can be constructed in two ways. Either one can construct it by concatenating the bit-representation of attribute values or one can use the calculation which is also used in programming languages to map multi-dimensional arrays to memory. Figure 3.1(a) on page 29 depicts the compound curve for a 8x8 universe, which is basically a row-wise scan.

Definition 3.9 (Compound SFC by calculation)
$C_c(\vec{p}) = p_1 + \sum_{i=2}^{d} (p_i \cdot \prod_{j=1}^{i-1} |\mathbb{D}_j|)$ and cardinality $|C_c| = \prod_{i=1}^{d} |\mathbb{D}_i|$ for $\vec{p} \in \Omega$. $\diamond$

Definition 3.10 (Compound SFC by concatenation)
$C_b(\vec{p}) = p_d \circ \cdots \circ p_1$ and cardinality $|C_b| = \prod_{i=1}^{d} 2^{\lceil log_2 |\mathbb{D}_i| \rceil}$ for $\vec{p} \in \Omega$. $\diamond$

The first mapping requires costly multiplications and when using types supported by a CPU one might easily generate overflows, e.g., the compound address of a two dimensional universe with 32 bit integers is 64 bit long. It also maps just to the domain $[0, |\mathbb{D}|[$ while the second mapping concatenates bit-strings and requires a larger domain $|C_b|$.

However, only $d - 1$ bits can be saved by using C_c instead of C_b, i.e., with b bits per attribute we have a minimum domain $[1, 2^{b-1} + 1]$ for each dimension and a maximum domain of $[1, 2^b]$. The resulting universe is the product of the domains of all the dimensions.

Proof 3.1 (Upper bound for additional bits required by concatenation of compound dimensions versus calculation)

$$
\begin{aligned}
log_2((2^b)^d) - log_2((2^{b-1}+1)^d) &= && (3.3)\\
d \cdot b \cdot log_2(2) - d \cdot log_2(2^{b-1}+\mathbf{1}) &< && (3.4)\\
d \cdot b \cdot 1 - (d \cdot log_2(2^{b-1})) &= && (3.5)\\
d \cdot b - (d \cdot (b-1) \cdot 1 &= && (3.6)\\
d \cdot b - (d \cdot (b-1)) &= && (3.7)\\
d \cdot (b - b + 1) &= d && (3.8)
\end{aligned}
$$

◇

The inverse function $C^{-1}()$ is easily calculated by reversing the calculation of $C()$ and thus has the same complexity.

The compound curve clusters data perfectly w.r. to p_1, but not w.r. to neighbors in other dimensions. For example, a partial match query restricting p_1 results in a single interval on the compound curve. However, there are $|D_1|$ points on the compound curve between two neighbors w.r. to dimension 2, $|D_1| \cdot |D_2|$ between two neighbors w.r. to dimension 3, thus dimensions are favored the nearer to the end of the key they are. There is no symmetry between dimensions w.r. to spatial clustering.

Furthermore, as it is a line-wise scan there is a jump between lines, i.e., there are neighbors on the curve which are not neighbors in Ω. This happens when moving from the last point w.r. to a dimension to the next one.

3.1.2 Snake-Curve and Zig-Zag-Curve

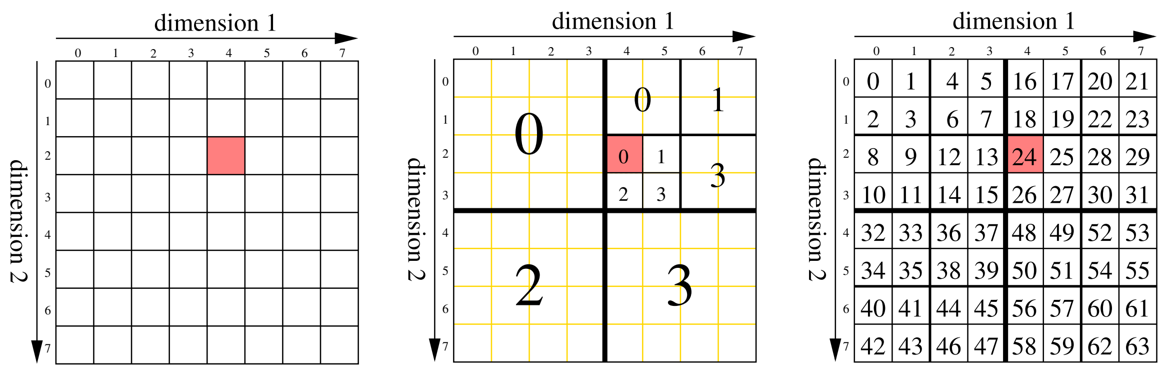
The jumps of the compound curve can be avoided by flipping the direction of the scan in each odd line, i.e., we scan one line forward and the next line backward (see Figure 3.1(b) and [Jag90a]). However, this does not fix the problem of favoring the clustering w.r. to the position of the dimension in the key.

Figure 3.1(c) depicts a Zig-Zag SFC which is a 45° rotated Snake-curve, which does not favor a dimension, but its clustering is depending on the position. For example, the four neighbors in the center of the universe $24, 31, 32, 39$ are on three segments of the SFC while the compound and Snake-curve have only two segments and thus do not fragment the data which is apparently not fragmented in the original universe.

In the following, curves with better symmetry and spatial clustering are discussed.

3.1.3 Fractal Curves

All the fractal curves presented here partition each dimension recursively into equal parts and then order the sub-cubes in their own specific way. For the two-dimensional case this



(a) Universe: 8x8 Grid (b) Partitioning (c) Addresses of Cells

Figure 3.3: Universe, quad tree partitioning, and address

partitioning is a quad-tree, in general it is a 2^d way tree, i.e., we have 2^d sub-cubes on each level.

The fractal SFCs differ in how they number the 2^d sub-cubes on each level of the tree. The SFC-address of a point can be expressed as path in the tree leading to this point. The path is denoted as concatenation of sub-cube numbers referring to the sub-cubes containing the point on each level of recursion.

The numbers assigned to the sub-cubes are in the range $[0, 2^d - 1]$ requiring d bits, and we need $s = log_2(|\mathbb{D}|)$ recursive levels in order to partition a universe Ω with $|\mathbb{D}| = |\mathbb{D}_1| = \cdots = |\mathbb{D}_d|$. Thus an address has the length $|\alpha| = n = s \cdot d$ bits. In [Mar99, Ram02] the term *step* is used instead of *level* of recursion resp. *level* of the tree. Following their conventions we will also call them *step* and separate steps by a ".". In related work *step* is also often called the *order* of the curve.

Example 3.1 (Different representations of addresses)

We assume an 8x8 universe with two dimensions having the domain $[0, 7]$ as depicted in Figure 3.3(a). Its quad-tree partitioning is depicted in Figure 3.3(b) where thicker lines indicate partitioning on a higher level. Only the second quadrant on the first level and the third quadrant on the second level are further subdivided in the figure. Sub-quadrants are numbered according to a row-wise scan.

There is a point indicated with the coordinates $(4, 2)$ and its address can be written as:

Concatenation of sub-cube numbers: q_i is the sub-cube number on level i and we write the address as $q_0.q_1.\cdots.q_{s-1}$

Integers: we get 1.2.0, i.e., on the first level the point $(4, 2)$ is included in the quadrant 1, on the second level in quadrant 2 and on the third level in quadrant 0.

Bit-strings: writing the sub-cube numbers as bit-strings we get 01.10.00

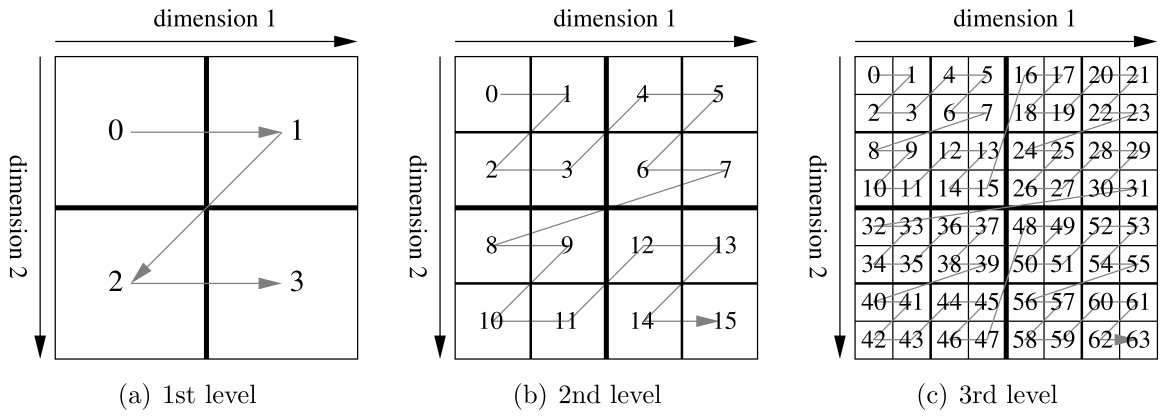


Figure 3.4: Z-curve for 8x8 universe

Address: We do not need the separating "." between steps and thus may write the address as $\alpha = \alpha_0 \cdots \alpha_n$ where step i has the value $q_i = \alpha_{i \cdot d+0}\alpha_{i \cdot d+1} \cdots \alpha_{i \cdot d+d-1}$ for $i \in [0, d-1]$:

Bit-string: Therefore, the address is the bit-string 011000

Integers: This corresponds to the integer 24 as in Figure 3.3(c).

The resulting addresses in number notation are depicted in Figure 3.3(c). $\diamond$

In the following different orderings for the sub-cubes are discussed and how the corresponding SFC addresses can be calculated.

3.1.3.1 The Z-curve

The Z-curve was introduced by [Mor66] and its calculation by bit-interleaving used for indexing was first proposed by [OM84]. It is a fractal SFC applying a "Z"-leitmotiv recursively to the sub-cubes, i.e., it uses the compound order on each level of partitioning.

Figure 3.4 depicts the recursive partitioning of a two dimensional universe for the first three recursions of the Z-curve, where the third level is the finest granularity, i.e., we have 8 discrete coordinates in each dimension in the range $[0, 7]$. The quadrants are numbered with the corresponding Z-address as an integer number and the cells are connected by a curve.

Obviously the Z-curve has jumps, since within a recursion level it is a compound curve. When comparing the Manhattan-distance of the two points $\Delta_M(\vec{p}_1, \vec{p}_2)$ with their distance on the Z-curve $\Delta_Z(\vec{p}_1, \vec{p}_2)$ two extremes can be observed violating the preservation of spatial proximity:

1. the Manhattan-distance is substantially smaller than the Z-distance, e.g., in Figure 3.4(b) for the two points $(0, 1)$ and $(0, 2)$ with Z-addresses 2 and 8 we have a $\Delta_Z = 6$ while $\Delta_M = 1$

2. on the other hand the Manhattan-distance is much greater than the Z-distance for the two points $(3, 1)$ and $(0, 2)$ with Z-addresses 7 and 8 we have a $\Delta_Z = 1$ while $\Delta_M = 4$

There are two ways to calculate Z-addresses: by concatenating the quadrant numbers of each recursive partitioning step containing a given coordinate (as described before) or by interleaving the bit-representation of the attribute values (Figure 3.5(a)). *Bit-interleaving* is more efficient since it does not require coordinate comparisons in order to check if a point is included in a sub-cube or not, but we shuffle bits. Furthermore, the one to one correspondence of bits in the address to bits in the dimension values allows for algorithms working only with addresses, i.e., coordinate comparisons can be performed solely on addresses and there is no need to calculate $S^{-1}()$.

Definition 3.11 (Z-curve by calculation)
$Z(\vec{p}) = bitinterleave(p_1, \cdots, p_d)$ ◇

The inverse function $Z^{-1}()$ is calculated by reversing the interleaving process and consequently it has the same complexity as $Z()$.

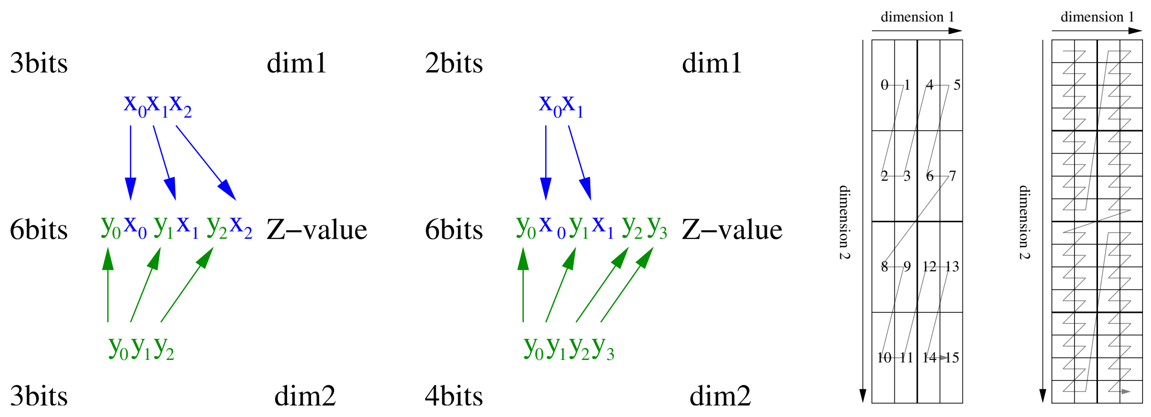
Bit-interleaving extends easily to any number of dimensions and is faster than the concatenation of quadrant numbers, since no comparisons in Ω are required to decide which quadrant contains a given point. Dimensions with differing cardinalities are supported by considering only those dimensions for bit-interleaving that have bits left for interleaving, i.e., if all bits of a dimension are exhausted it will not be considered for the interleaving process anymore. Figure 3.5(b) depicts this for a 4x16 universe and 3.5(c) and 3.5(d) show the resulting curve on different levels.

Starting the interleaving with dimension d leads to the Z-shape while starting with dimension 1 would result in a N-shape, thus we always interleave them starting with dimension d. As the Z-curve is not perfectly symmetrical w.r. to all dimensions within a step, it is left to the user to specify the dimensions in the desired order.

Example 3.2 (Different representations of a Z-address)
For an 8x8 universe with the dimensions x and y and the corresponding bit-strings $x_1x_2x_3$ and $y_1y_2y_3$ we can write the Z-address as bit-string $y_1x_1y_2x_2y_3x_3$. We may add an "." to separate steps and write the address as $y_1x_1.y_2x_2.y_3x_3$ or $q_0.q_1.q_2$ where q_i denotes the quadrant on recursion level i containing the corresponding point. ◇

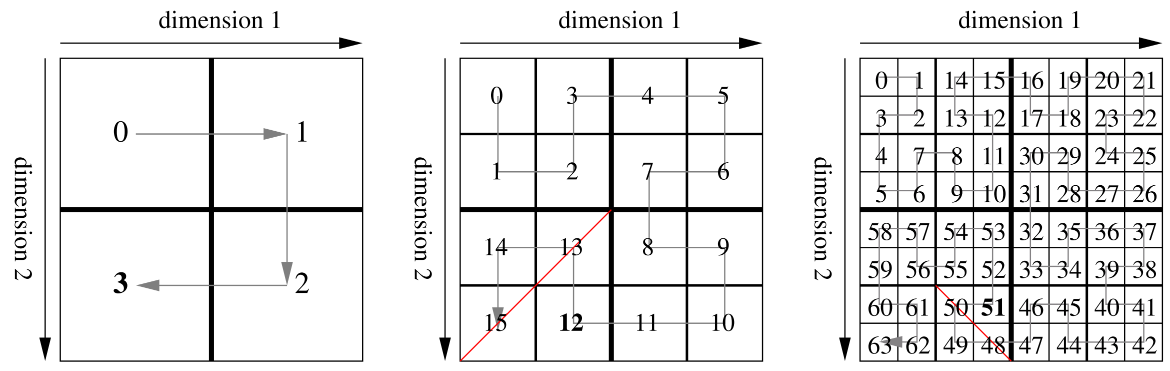
The calculation can be optimized by look-up-tables encoding what address bit corresponds to what bit of a dimension. The complexity of the calculation is $O(2|\alpha|)$, i.e., it is linear to the length of the address requiring to read and write each bit once.

Due to the compound order within each recursion step of the Z-curve it is still favoring dimensions w.r. to their position in the key, but this happens only within one step. By applying the idea of the Snake-curve to each recursion step we can avoid the jumps within a quadrant, but not those between quadrants. In order to avoid jumps between quadrants, they have to be rotated to connect at their end and start points. This is the basic idea of the next curve, the Hilbert-curve.



(a) 8x8 (b) 4x16 (c) 2nd (d) 4th

Figure 3.5: bit-interleaving for dimensions with differing domains



(a) 1st level (b) 2nd level (c) 3rd level

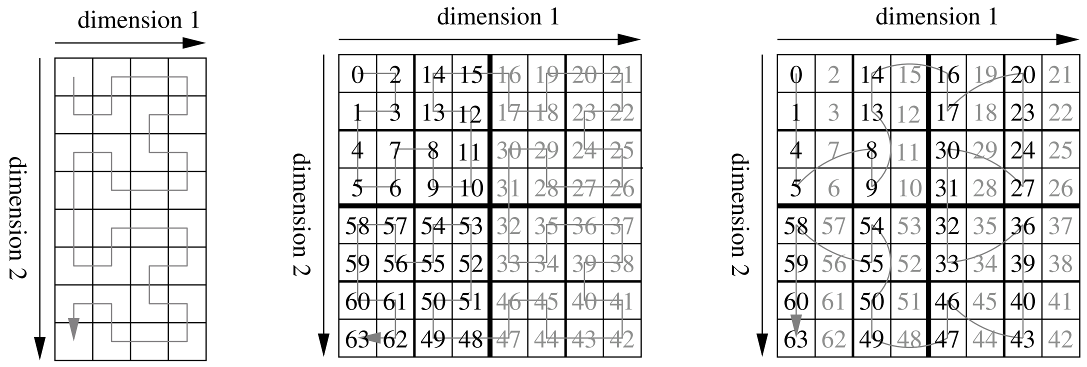
Figure 3.6: Hilbert-curve for 8x8 universe

3.1.3.2 The Hilbert-curve

The Hilbert-curve is a fractal curve introduced by [Hil91] for the two dimensional case, but it can be generalized to any number of dimensions.

Its basic leitmotiv is a “U” which is rotated and flipped during the recursive partitioning in order to generate a curve without jumps. Rotating and flipping is done in dependency of the orientation of the father step. Figure 3.6 shows the Hilbert-curve for a two dimensional universe.

While it is easy to hard-code the generation for the 2d and 3d space it becomes tedious to do it for higher dimensional spaces. [But71] presented a generic algorithm to calculate the Hilbert-address for an arbitrary number of dimensions and thus eliminated the need for hard coded algorithms. [Moo00] has slightly enhanced the algorithm to be more efficient. [Law00] proposes compressed state diagrams to speed up the calculation for up to 8 dimensions and reduce its complexity to $O(|\alpha|)$. Compression is necessary, since storing all states, i.e., the whole quad-tree, introduces huge memory requirements. He generates the



(a) Modified Hilbert (b) Direct embedding (c) Padding dimensions

Figure 3.7: Hilbert-curve indexing an universe with differing dimension cardinalities

state-diagrams required for an instance of the Hilbert-curve with the algorithm of [But71].

Basically, the calculation of the Hilbert address works as follows; first the Z-address of a point is calculated then the bits of a step are modified to form the "U" shape which is accomplished by XORing the bits of a step with all higher order bits of the step. Now the "U"s have to be rotated and flipped in order to be connected at their end and start points. This has to be done depending on the quadrant number within a step and the orientation of the father quadrant.

The inverse function is calculated by reversing this process.

Example 3.3 (Hilbert Address Calculation)

For an 8x8 universe with the Z-address $y_1x_1.y_2x_2.y_3x_3$ we get the "U"s by XORing it with $0y_1.0y_2.0y_3$, e.g. point $(3, 6)$ has the Z-address $10.11.01 = 2.3.1$ which becomes $10.11.01$ *xor* $01.01.00 = 11.10.01 = 3.2.1$. Now the quadrants have to be flipped diagonally that corresponding to start (quadrant number 0) and end (quadrant number 3) points to connect to the other "U"s. The quadrant on the first step containing the point, has number 3, thus sub-quadrants 0 and 2 we have to be swapped (Figure 3.6(b) shows the flip diagonal) and also apply this to the following steps, i.e. we get 3.0.1. The second step is now 0 and we have to apply another flip, this time by the main diagonal and finally we get 3.0.3 which is 51 (Figure 3.6(c)). $\diamond$

The Hilbert-curve requires a universe with dimensions of equal cardinality in order to maintain its property to have no jumps. Although there are curves without jumps (e.g., Figure 3.7(a)) to the best of our knowledge there is no algorithm published so far to calculate these SFC for an arbitrary universe.

Thus universes with differing cardinalities can be managed by embedding them into an universe where all dimensions have the maximum cardinality, but it is no longer guaranteed to be without jumps. For example, 8x4 universe can be embedded without jumps and using only the necessary bits when using the same origin, but this does not work for a 4x8 universe unless it is rotated, i.e., the order of dimensions. There appears a huge jump from 15 to 48 as depicted in Figure 3.7(b). Furthermore, all addresses share a common prefix of either 00 or 11. Due to this, this embedding should reorder the dimensions in descending cardinality of dimensions.

In order to eliminate the common prefixes we can pad the bit-strings of the attributes to the length of the maximum attribute length by appending 0s before calculating the Z-address. This basically is an embedding of dimensions with smaller cardinality on a coarser level of recursion. Unfortunately this introduces additional jumps (Figure 3.7(c)), but those occur only on the level of finest granularity and thus have a minor impact as those points a likely on the same page.

3.1.3.3 The Gray-Code-curve

[Fal86] has proposed to use Gray-codes [Gra47] for hashing of multi-attribute data to efficiently support partial match queries. The Gray-Code-curve can be calculated by encoding the dimension values according to a Gray-code, i.e., the encoding of successive dimension values differs only in one bit position which means the Hamming distance is 1. The Gray code is efficiently calculated by xoring the dimension value with itself shifted by 1 ($\gg 1$) to the right. Then applying bit-interleaving results in a multidimensional Gray code.

Definition 3.12 (Gray-Code-curve calculation)
$G(\vec{p}) = bitinterleave(p_1 \ xor \ (p_1 \gg 1), \cdots, p_d \ xor \ (p_d \gg 1))$ ◇

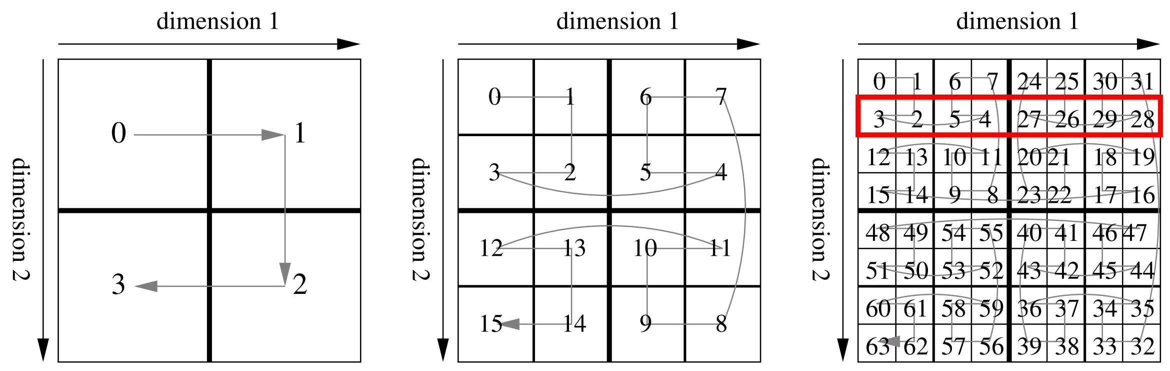
The inverse function cannot be calculated that easily, since we would need the shifted value for another XOR, but this value is lost during encoding. We may calculate the value by XORing each bit of the Gray-code with its higher order bits, see [Law00] for an algorithm which performs this in linear time w.r. to the address length.

The curve is depicted in Figure 3.8 and its distinct property is that all jumps are axis parallel. Due to this, it may have fewer segments for a partial match query than the Hilbert-curve or Z-curve. For example, the partial match query restricting dimension 2 to 1 (Figure 3.8(c)) results in two segments on the Gray-Code-curve while there are four segments for the Hilbert-curve and Z-curve.

3.1.4 Curves created by order preserving Bit-Permutations

As clustering is a crucial point for query performance, one might want to choose the SFC providing the best clustering for a given query profile. [Ram02] proposes a set of SFCs created by weighted interleaving of the dimensions. Instead of taking one bit from each dimension in round-robin manner he allows to take several bits from one dimension at once. Thus the mapping is still order preserving, i.e., higher dimension values result in a higher address, but the resulting clustering is modified. All possible curves have jumps, but their calculation and the calculation of the inverse is efficiently done by bit shuffling.

The Z-curve and the compound curve are special cases of this class of curves, where we are taking bits in round robin manner for the Z-curve and all bits of a dimension at once for the compound curve.



(a) 1st level (b) 2nd level (c) 3rd level

Figure 3.8: Gray-Code-curve for 8x8 universe

[Ram02] also discusses how to make a decision on the right curve with respect to a given application resp. query profile. The main idea is to favor the dimensions in the clustering process which lead to a higher selectivity. A dimension is preferred by assigning more higher order bits (bits at the start of the address) to it. The compound curve is one extreme clearly showing how a specific query class can be supported most efficiently, e.g., if we are restricting dimension two, a compound key on $dim2 \circ dim1$ is best, since it will always result in one segment on the curve as one can see in Figure 3.1(a) on page 29.

3.2 Indexing

For query processing it is required to map the query on the multidimensional space to a query on the SFC. Multidimensional range queries map into a set of interval queries on the one-dimensional SFC. In order to accomlish this the points resp. addresses where the SFC is entering and leaving the query-shape have to be calculated. Specifically the following functions are required to perform this mapping:

$S(\vec{p})$: maps a tuple $\vec{p} \in \Omega$ to its address in S.

$S^{-1}(\vec{p})$: maps a SFC-address to they key part of a tuple.

$min_S(Q)$: calculates minimum address within the query shape Q.

$max_S(Q)$: calculates maximum address within the query shape Q.

$NJI(\vec{p}, Q)$: calculates the address of the next point $\vec{q}$ (the **N**ext **J**ump **I**n) with the smallest address greater than $\vec{p}$ which is within the query box Q.

$NJO(\vec{p}, Q)$: calculates the address of the next point $\vec{q}$ (the **N**ext **J**ump **O**ut) with the smallest address greater than $\vec{p}$ which is just before the SFC leaves the query box Q again.

These functions are not limited to a specific query shape, however efficient implementations are usually restricted to range queries, i.e., Q specifies one interval restriction on each dimension. An implementation might just require a sub-set of these functions, i.e., S^{-1} might not be necessary.

A query shape maps to a set of segments on the SFC. In order to obtain all segments, we start with $cur = min_S(Q)$ which is the first point within the query shape and then alternately call $cur = NJI(cur)$ and $cur = NJO(cur)$ until we get a cur equal to $max_S(Q)$.

A query might map to as many segments as points it contains, when there is one or more points on the SFC between two successive points within the query shape. Thus it is crucial that the NJI and NJO can be calculated efficiently, i.e., in linear time w.r. to the address length or better. On the other hand, $min_S(Q)$ and $max_S(Q)$ need to be calculated just once, but still an efficient calculation is desired.

In the following the SFCs are discussed again w.r. to these algorithms.

3.2.1 Address calculations

The addresses $min_S(Q)$ and $max_S(Q)$, which must not exist as actual points in the table, are formally defined as:

Definition 3.13 ($min_S(Q)$)
$min_S(Q) = min(\{S(\vec{p})|\vec{p} \in Q\})$. ◇

Definition 3.14 ($max_S(Q)$)
$max_S(Q) = max(\{S(\vec{p})|\vec{p} \in Q\})$. ◇

While they are easily defined as the minimum resp. maximum address of the points within a query shape we definitely cannot afford to calculate them by calculating the minimum resp. maximum address of all points within the query. The same holds for the NJI and NJO.

Formally we define the NJI and NJO conforming to [Mar99, Ram02] as:

Definition 3.15 **(Next Jump In,** $NJI(\vec{q}, Q), nji$**)**
The next jump in w.r. to a given point $\vec{c}$ and a query Q is defined as $NJI(\vec{c}, Q) = min(\{S(\vec{n})|\vec{n} \in Q \wedge S(\vec{c}) < S(\vec{n})\}$ for $min_S(Q) \leq S(\vec{c}) \leq max_S(Q)$. ◇

Definition 3.16 **(Next Jump Out,** $NJO(\vec{q}, Q), njo$**)**
The next jump out w.r. to a given point $\vec{p}$ and a query Q is defined as $NJO(\vec{p}, Q) = min(\{S(\vec{q})|\vec{q} \in Q \wedge S^{-1}(S(\vec{q}) + 1) \notin Q\}$ for $\vec{p} \in Q \wedge min_S(Q) \leq S(\vec{p}) \leq max_S(Q)$ ◇

While we are using a point $\vec{c}$ as input here, we may also use an address where appropriate, i.e., when working just with addresses during query processing we can avoid calls to S and/or S^{-1}. The NJI-algorithm returns a point greater than the input point, i.e., $S(\vec{c}) < NJI(\vec{c}, Q) \leq max_s(Q)$. The NJO-algorithm may also return the point itself, i.e., $S(\vec{c}) \leq NJO(\vec{c}, Q) \leq max_s(Q)$.

In the following we will examine the algorithms and their cost for the calculation of addresses for different SFCs and a range query $Q = [\vec{l}, \vec{u}]$.

3.2.1.1 Compound-curve and Snake-curve

The compound curve is *dimension order preserving* according to Definition 3.6 on page 31 and thus we can obtain the minimum resp. maximum address within the query shape as $min_S = S(\vec{l})$ resp. $max_S = S(\vec{u})$.

The compound curve can be seen as a tree of height d with branching degree of $|\mathbb{D}_i|$ on level i. The NJI and NJO can be calculated in linear time w.r. to the number of dimensions. The calculations described here operate on dimension values, but as the address gets constructed by concatenating them we can easily adapt them to also work on addresses.

The NJI is calculated by, scanning from the start of the address to its end, we search for the last dimension which can still be incremented, i.e., its value q_i is within $[l_i, u_i - 1]$. We then increment this value and set all remaining ones to their minimum w.r. to the query, i.e., $q_j := l_j$.

This algorithm requires to scan the address once. [Ram02] presents a similar algorithm called *skipper* and analyses its performance.

It should be noted that practically all implementations of B-Trees supporting compound keys do not utilize the decomposition into smaller sub-intervals, but they simply fetch the whole range

$$[SFCMin(Q), SFCMax(Q)] \tag{3.9}$$

and apply post filtering to the results (see Figure 2.7(b) on page 27). This stems from the fact that the original B-Tree range query has been designed for single interval queries, but not for a set of simultaneous interval queries.

The NJO can be calculated for an input which is a nji as follows. Starting with the last dimension in the address it is set to its maximum, i.e., $q_1 := u_1$. If q_i covers the whole dimension or there is no restriction on dimension i, we continue with the preceding dimension otherwise or if no dimension is left we stop.

For the snake scan the NJI and NJO calculations are performed in a similar way taking into account that we are reversing the direction with every even dimension value. Thus instead of setting dimension values always to l_i resp. u_i in the NJI and NJO calculation we set it to their u_i resp. l_i for even dimension values. In case of the NJI we also have to decrement instead of incrementing for even dimension values and thus additionally have to search for the last position where a decrement is allowed.

3.2.1.2 Fractal Curves

For *dimension order preserving* fractal curve the minimum and maximum addresses within the query shape are $min_S = S(\vec{l})$ and $max_S = S(\vec{u})$. This is true for the Z-curve, but not for the Gray-Code-curve and Hilbert-curve, since a higher dimension value does not guarantee a higher address. Furthermore, these points may not be located at the corners of the range query but somewhere else on its outline.

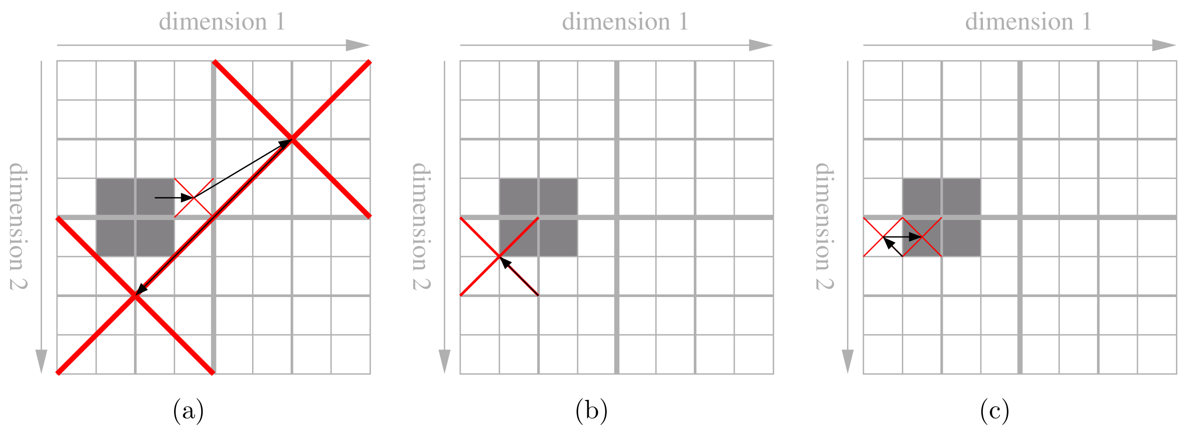


Figure 3.9: Brother-father-search with Z-curve

In order to find min_S resp. max_S we have to descend the tree representation starting at its root along the path with the lowest resp. highest sub-cube number intersecting the range query. The tree is actually nowhere stored, but Z-addresses represent an encoded version of the tree and can be used for this traversal. Starting with the origin $\vec{p} = (0, \cdots, 0)$, the sub-cube coordinates $p_1, \cdots, p_d$ can be calculated by $p_i = \sum_{j=1}^{s} 2^{s-j} \cdot b(i, q_j)$, where $b(i, q_j)$ is 1 if the bit of dimension i is set in the sub-cube number q_j. This is done iteratively while descending the tree. The decision for the path of the further descend is controlled by the sub-cube numbers specific to the used curve. These sub-cube numbers can be obtained from the calculation of $S()$.

Thus s iterations are required, where s is the height of the tree, and 2^d coordinate calculations and comparisons on each level leading to complexity of $O(s \cdot 2^d)$.

A generic algorithm to calculate the NJI resp. NJO for all fractal curves is a brother/father search in the underlying quad-tree in the order of the curve. The term *brother* refers to sub-cubes on the same level within tree node, while father refers to the node pointing to the current node.

Search for the next older bother intersecting the query shape starts at the leaf level of the tree where the starting point of the search is located. Older brothers are those with higher sub-cube numbers. If there is no match one goes to the father and search all older brothers w.r. to the father level. This continues until finding a match or no older brothers resp. fathers are left.

Instead of traversing the tree, which is never materialized, we again utilize the address representation $\alpha = q_0.q_1.\cdots.q_{s-1}$ for traversal. Moving to the next older brother on level i is done by incrementing the sub-cube number of that level, i.e., $\alpha.q_i = \alpha.q_i + 1$. If the sub-cube number has reached the maximum $2^d - 1$ we set it to 0 and increment the sub-cube number of the previous level (father level) $q_{i-1} + 1$.

[Bay96] proposed this algorithm for the Z-curve. The number of brothers to test for intersection is $2^d - 1$ on each level in worst case and we have to test on s levels in worst case, thus we have an exponential complexity with respect to the number of dimensions of

$O(s \cdot (2^d - 1))$. Furthermore, the intersection test has to be performed on the coordinates in Ω and thus we have to calculate these coordinates from the addresses. In case of the Z-curve the comparison can be reduced to one per dimension, by comparing with the partitioning value w.r. to the dimension and thus get a linear complexity of $O(s \cdot d)$.

Working solely on addresses avoids comparisons of coordinates resulting in a significant performance gain as discussed in [Fri97, Mar99]. An algorithm developed by Volker Markl is presented and explained in detail in [Ram02]. It works by comparing the current Z-address with $S_{Zmin}(Q)$ and $S_{Zmax}(Q)$ detecting what dimensions fall below the queries lower bound l or exceed the queries upper bound u and thus know where to increment and correct the address in order to gain the nji or njo. [Ram02] presents detailed complexity discussions and also a NJO as they are used in the UB-Tree implementation. Furthermore, [Ram02] shows that these algorithms also work for other curves created by order-preserving bit-interleaving.

Also [TH81, Law00] present similar algorithms for the NJI calculation on the Z-curve.

The Hilbert-curve algorithms for the NJI calculation are all based on the tree traversal as described before but with different representations encoding the underlying tree. [Law00] presents a NJI based on state-diagrams for the tree traversal. Recently [Tro03] provides a NJI algorithm based on addresses.

On the Gray-Code-curve there do not exist publications on the NJI and NJO algorithms, but as the tree traversal does not dependent on actual curve it can also be utilized here. In fact, the curve just determines the order in which sub-cubes are checked.

When tracking the orientation of the "U" in the Hilbert-curve and Gray-Code-curve we may also reduce the number of comparisons to one comparison per dimension by just comparing with the partitioning value w.r. to the dimension and thus getting a linear complexity of $O(s \cdot d)$. However, this requires additional book keeping of the orientations and corresponding data structures.

3.2.2 Indexing a Space Filling Curve

A SFC maps points of a multidimensional space to points in a one dimensional space. As there are well known and stable indexing solutions for one dimensional space, we are now able to utilize such an index also for multidimensional problems and benefit from the efforts and developments done for the one-dimensional index.

Instead of indexing the dimensions with a single primary compound index or multiple secondary indexes, we index the SFC-address of the tuples with a one dimensional clustering index, e.g., a B-Tree. Tuples are now spatially clustered according to the SFC. The SFC-address can be calculated on the fly if functional indexes are supported or might be stored as an additional column in the relation.

This approach requires adding or modifying application code in order to take the SFC into account for query processing as well as for data management. [OM84] has first presented this technique in combination with the Z-curve.

Depending on the DBMS the SFC-support might be incorporated into the DBMS by proprietary interfaces, e.g., programmable SQL, cartridge, data blades, etc.. This reduces the code on the application side and makes the SFC technique also easier available to other applications. For this purpose, [BKK99] have utilized Oracles programmable SQL.

However, a SFC-function itself is not sufficient in order to provide support for multidimensional queries, i.e., range queries, sorted reading, nearest neighbor search. For example, a multidimensional range query maps into a set of intervals on the SFC, see the query box in Figure 3.1. Consequently we have to process a set of interval queries on the one-dimensional index. In case of page clustering, processing the intervals in SFC-order makes optimal use of prefetching.

Calculating the SFC-intervals for a query box in advance causes extra query processing cost due to the following reasons:

1. The number of intervals can get quite high, thus exceeding the maximum query statement size or DBMS optimizer capabilities.

2. Query processing is blocked until all intervals are calculated.

3. The real data distribution is neglected and thus there is a high probability that most of the calculated intervals are empty in sparsely populated universes.

One can reduce the number of intervals by calculating them on a coarser granularity, but this also leads to false positives, i.e., tuples that are not part of the result, and therefore requires post filtering.

Therefore, a cursor driven approach calculating the necessary intervals based on the real data stored is a better solution. A cursor based algorithm requires only the following four functions: $S(\vec{p})$, $min_S(Q)$, $max_S(Q)$ and $NJI(\vec{q}, Q)$.

Range query processing works as follows: Starting with the minimal address we read tuples from the index as long as they are within the query box, i.e., we perform an interval query $[min_S(q), max_S(q)]$. All those tuples within the query box are result tuples which can be pipelined for further processing. When encountering a tuple t outside of the query box we close the cursor, calculate the nji and open a new cursor with the interval query $[NJI(t, q), max_S(q)]$. This is continued until there are no more tuples left. This algorithm is listed in Table 3.1.

Thus we skip the calculation of all those intervals between two consecutive tuples which are both within the query box. Although most DBSMs provide caching and a way to avoid re-parsing and re-optimizing the statement, we usually will access data pages multiple times and thus generate additional cost for query processing.

The size of the interval queries can be reduced by replacing $max_S(q)$ with

$$NJO((NJI(t, q), q))$$

and proceed by calculating the nji for the last njo. This is an iterative version for processing all the intervals a query box decomposes into. Thus we can avoid post-filtering, but likely we are adding additional computations of nji and njo compared to the method above. In the worst case, they have to be calculated for each tuple obtained. We may

```
void Application::RangeQuery(Pipeline &pipe, const Query &Q)
{
  Cursor cursor(Range(S_min(Q),S_max(Q)));
  while (cursor.moreTuples() == true) {
    Tuple tuple = cursor.fetchTuple();
    if (tuple.withIn(Q)) {
      pushTuple(pipe);
    } else {
      cursor.reopen(Range(NJI(S(tuple)),S_max(Q)));
    }
  }
}
```

Table 3.1: Generic cursor based SFC range query algorithm

switch to the algorithm described above if there is no result in order to obtain a better starting point for the nji by honoring the data actually stored.

The NJI and NJO algorithms are usually called frequently and thus their performance is crucial. Their complexity should be linear to the address length and they should be memory based in order to be suitable for efficient indexing.

The cost for query processing can be measured by the number of calls to NJI and NJO and the number of accessed pages. With this approach we usually have a significantly higher number of NJI calculation than pages retrieved. Furthermore, the search within a page is not honoring searches on that page done before, i.e., for each nji falling on a page we have to search the whole page again. Current DBMSs are not handling this well, as each search is a query by itself they are processing the whole page again as they do not know about the dependency of the queries, i.e., that addresses are increasing.

The algorithms above are limited to range queries. To process other query types usually other specialized address calculations and processing strategies are necessary, e.g., tetris [MZB99], nearest neighbor [Mar99], skyline [KRR02], but they are not discussed in this thesis.

3.2.3 SFC Index

Integrating a SFC into a DBMS kernel providing a B-Tree variant allows for various optimizations in order to gain the best performance while reducing modifications within application code to a minimum. When speaking of B-Trees we refer to the B$^+$-Tree, where data is stored on the leaf pages. We assume items to be stored according to the address order on both, the index and data pages.

The optimizations to be considered are:

1. Page based access pattern,
2. freedom to choose the split addresses (the keys in the index part),

```
void SFCIndex::RangeQuery(Pipeline &pipe, const Query &Q)
{
  Address nji = S_min(Q);
  while (1) {
    DataPage page = fetchPage(nji);
    Address njo = NJO(nji);
    if (nji <= S(page.first()) && S(page.last()) <= njo) {
      pipe.push(page.content());
    } else {
      pipe.push(filterTuples(Q, page.content()));
    }
    if (page.separator >= S_max(Q)) break;
    nji = NJI(page.separator);
  }
}
```

Table 3.2: Generic page based SFC range query algorithm

3. optimized tree traversal,

4. avoiding post filtering,

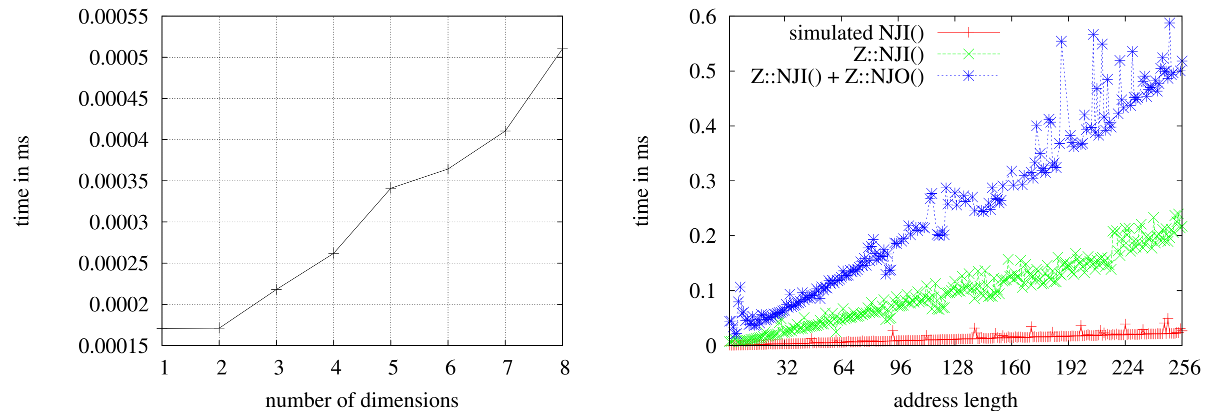
5. optimizer tuning,

6. reduce application code modifications to DLL statements.

When indexing the last address within a region with a B-Tree we implicitly define a partitioning of the universe into a set of disjoint intervals on the SFC. Each SFC interval corresponds to a region in Ω and its data is stored on a data page. The partitioning is defined by the split-addresses stored in the B-Tree. Thus we can switch from tuple based access driven by a cursor to a page based access. This avoids multiple accesses to the same page and minimizes the processing of the page to a single pass over all tuples. We have to fetch only those pages corresponding to regions intersecting the query box. The nji and njo are now calculated once for each accessed page, where the njo is used to identify regions which are completely within the query box and thus do not require post-filtering.

The corresponding algorithm is listed in Table 3.2 and its cost for range query processing can be measured by the number of calls to NJI and NJO or the number of accessed pages, which is equal in this case.

When comparing the cost of post-filtering versus the calculation of the NJI and NJO we have to take into account that for a containment comparison, e.g., on an integer interval, we require a loop over all dimensions and at most two CPU instructions for the comparison of the current value with the lower and upper boundary on this dimension, thus we get a complexity of $O(2d)$ instructions. On the other hand, calculating the nji or njo in best case is linear to the number of bits $n = |\alpha|$ of the address, and thus, i.e., it requires reading n



(a) Post filtering (b) NJI/NJO calculation

Figure 3.10: Cost of post filtering vs. NJI/NJO calculation

bits from the current address, $S_{min}(Q)$ and $S_{max}(Q)$ by offset calculation and bit-masking. Bits from the current address are compared against those of the query bounds. Finally up to n bits in the new address have to be modified, again requiring offset calculation and bit-masking. When assuming one instruction for each of these steps we get $O(6n)$. Therefore, post-filtering depends just on the number of dimensions, but nji and njo calculation on the number of bits of the address and thus the volume of the universe $|\Omega|$.

Figure 3.10 shows the times measured for one call to the post-filtering function, the simulated calculation of the NJI and NJO as described above and our implementation of the NJI and NJO for the UB-Tree. The time for one call was calculated as the average of 1000 calls to these functions on a Pentium 400. Post-filtering is ≈ 60 times faster compared to the simulated NJI and NJO as described above, but ≈ 500 times faster compared to our C++ based NJI for the Z-curve and when adding the time for the NJO we are ≈ 1200 times faster with post-filtering. Thus post-filtering data pages is usually superior to NJI and NJO calculations for filtering the tuples on the data pages, especially when we do not store the addresses of tuples on the data pages, since this would require additional calculation of the addresses of all tuples on the page.

Furthermore, we have the freedom to choose the split addresses. With a cursor based approach the decision for the split address is left to the implementation of the underlying one-dimensional index. These usually do not take the special access pattern and multidimensional partitioning of SFCs into account and therefore choose suboptimal split addresses. In order to optimally support range queries we want to create rectangular regions with as few fringes ([Mar99]) as possible. Figure 3.11 on page 50 depicts this for the Z-curve and the Hilbert-curve.

The best split address is the one keeping fringes as low as possible, i.e., keeping regions as rectangular as possible. This has been analyzed for the Z-curve in [Fri97, Mar99, Ram02]. The calculation of the best split position presented in [Mar99] actually works for all fractal curves. As split address we choose the common prefix of the last tuple on the first half and the first tuple on the second half filled-up with 1s resp. 0s depending on whether the separator is part of the region or not.

Optimized B-Tree traversal and search is now possible by caching the path and the position within the index page to the last accessed data page. Instead of always starting a new key search at the root of the index we continue traversal of the index at the position of the last match, i.e., the leaf index page. This reduces the number of accessed index pages significantly, i.e., we will fetch all index pages just once instead of multiple times where they are leading to multiple data pages intersecting the query. Furthermore, within the index page we are just searching in the remaining range of addresses. Thus both I/O and CPU cost are reduced. Figure 3.12 depicts this. If the index pages on the same level are linked to each other it is also possible to directly go where appropriate to the next index page on the same level.

Unnecessary post filtering can be avoided by checking if the whole content of a data page is within the range query. If the start and end address of an region is within the interval $[nji, njo]$ (see [Ram02]) post filtering is not necessary. Taking the page content into account allows for even better decisions, i.e., if the address of the first and last tuple are within $[nji, njo]$. In this case, evaluation of the range query predicate on all tuples is no longer necessary and thus all tuples starting from that position on the page can be directly pushed into the pipeline for further processing.

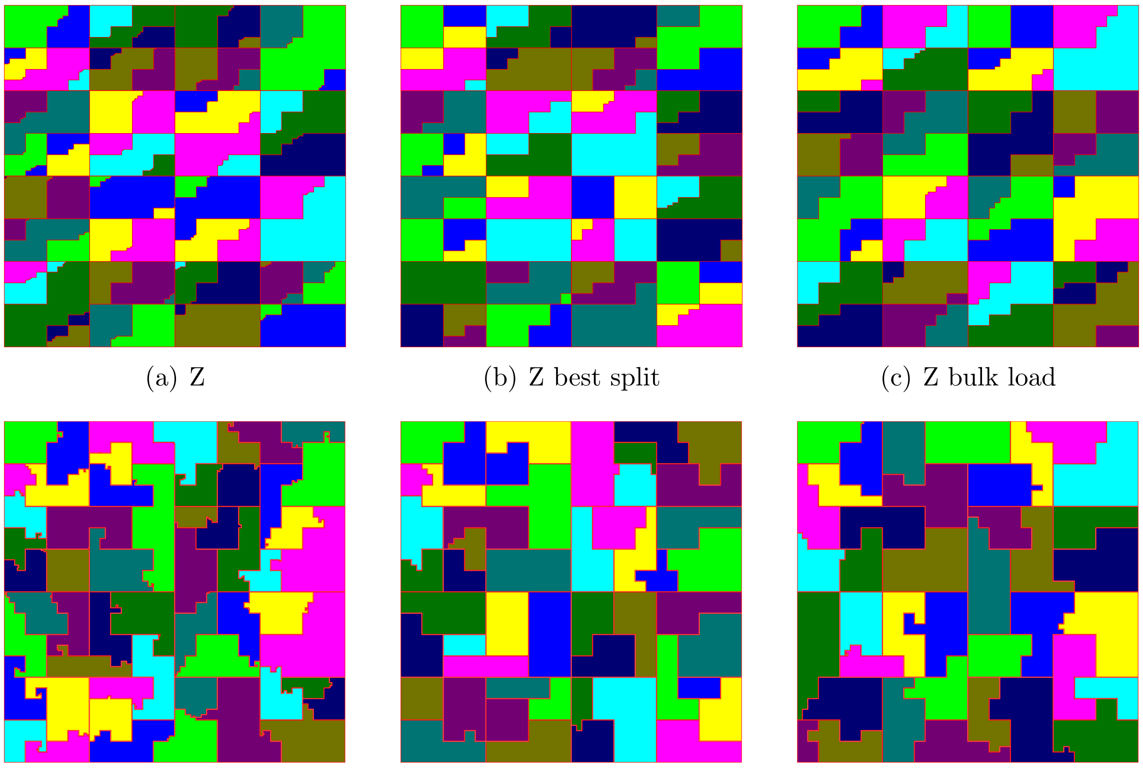
Optimizer extensions, e.g., cost function, statistics cardinality, take the properties of the SFC into account and allow for the decision of an optimal processing strategy. Without integrating a SFC into the DBMS kernel optimizers lack insight knowledge about the cost functions and supported query patterns and thus may decide for suboptimal execution plans.

Application code requires changes in order to take new index types into account. With an integrated index these modifications are reduced to modified DDL (Data Definition Language) statements, i.e., the SLQ `CREATE INDEX` statement. In general, code responsible to query processing is not affected anymore and thus the modifications are reduced to a minimum.

3.3 Related Work

In this section we will discuss results of related work on the comparison of SFCs. At the end of the section we will provide our perspective on the presented results. While SFCs had been discovered more than one century ago they came into the focus of researchers in the recent decades due to the emerging of multidimensional applications with huge data volumes that ought to be managed and queried efficiently.

Most of the analysis have only covered two and three-dimensional universes and focus on the three widely accepted recursive SFCs, Z-curve, Gray-Code-curve, and Hilbert-curve. The related work presented here is a small subset of the related work on that topic. We are considering commonly cited papers, work that is not restricted to 2d universe and some recent work. Following the references in the papers we are discussing here the reader will



(a) Z (b) Z best split (c) Z bulk load

(d) Hilbert (e) Hilbert best split (f) Hilbert bulk load

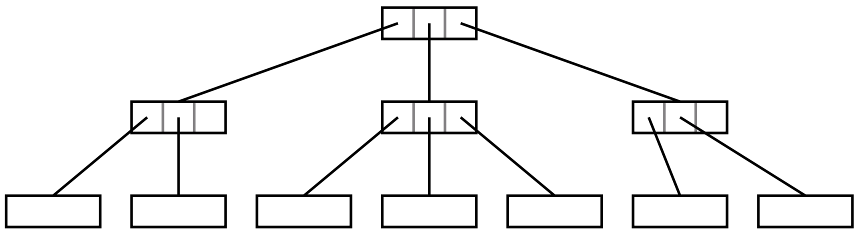
Figure 3.11: Space partitioning for Z- and Hilbert-curve

The split address is chosen as follows:

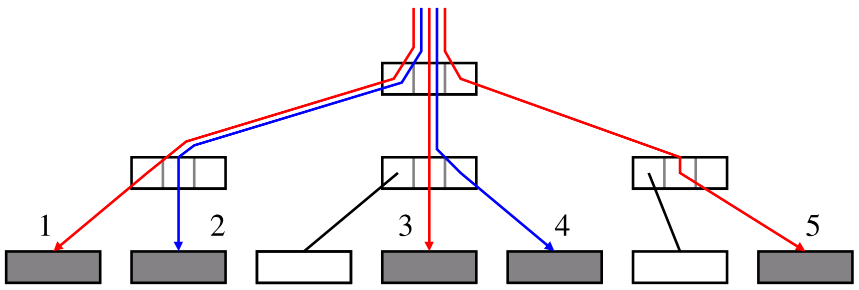
Figure 3.11(a) and (d): taking the address of the last tuple on the first half of the page

Figure 3.11(b) and (e): the best split position (the shortest separator) between the last tuple on the first half and the first tuple on the second half during random insertion

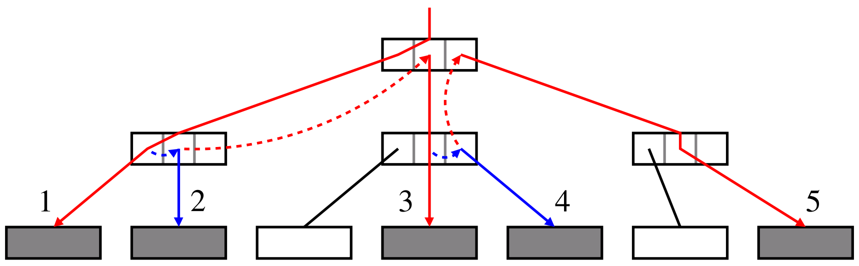
Figure 3.11(c) and (f): the best split position between the last tuple of page and the first tuple on the next page during bulk loading



(a) B-Tree Structure



(b) Traversal starting from the root of the B-Tree: 15 page accesses, i.e., three for each accessed data page



(c) Traversal continuing at last match: 9 page accesses

Figure 3.12: B-Tree traversal: Page accesses

find all relevant papers on this topic.

3.3.1 [Jag90a]

Linear Clustering of Objects with Multiple Attributes

[Jag90a] discusses and analyzes the properties of the Z-curve, Hilbert-curve, and Gray-Code-curve for the two dimensional case. In his analysis he considers partial match and range queries. As metric he uses the average number of non-consecutive intervals and the size of the linear span for a given selection $(max_S(Q) - min_S(Q))$. As cardinality $|\mathbb{D}|$ he assumes 2^m where m is the number of bits required to represent the values in $\mathbb{D}$.

Partial Match Queries: one dimension is restricted to a point the other dimension is unrestricted.

- **Compound-curve, Snake-curve:** For horizontal restrictions, i.e., dimension 1 is not restricted and dimension 2 is restricted to a point. This is clearly the best choice since we get one continuous run. However, when swapping the restrictions we cut the SFC at every grid cell like cutting a puff pastry and need 2^m runs (see Figure 3.1).
- **Z-curve:** For horizontal restrictions each pair of points is connected thus $\frac{2^m}{2}$ runs are required. For vertical restrictions we need 2^m runs. In average we get $1.5 \cdot 2^{m-1}$ runs.
- **Gray-Code-curve/Hilbert-curve:** These two curves also have connected segments in the vertical dimension and thus require fewer runs in the vertical dimension.

Summarizing the Z-curve requires $\frac{3}{4}$ more runs than the other SFCs which require in average $|2^{m-1} + 2^{-m-1}|$.

Range Queries: He simplifies the analysis to 2x2 range queries, i.e., two grid units in each dimension.

- **Compound-curve, Snake-curve:** With the compound curve we always require 2 runs, while the Snake-curve may have only one run when the query range is aligned to the turn of the Snake-curve.
- **Z-curve:** If the query range aligns to an even boundary in both dimensions it requires one run. For other positions we require more runs with a maximum of 4 runs when the query is centered. In average we get 2.625 runs.
- **Gray-Code-curve:** Aligned queries again require one run, summing up the other runs we finally get 2.5 runs in average.
- **Hilbert-curve:** Aligned queries again require one run, summing up the other runs we finally get 2 runs in average.

With this metric the Compound-curve, Snake-curve and Hilbert-curve are the best, followed by the Gray-Code-curve and finally the Z-curve.

With respect to the linear span there was no significant difference between the SFCs. Thus the analysis could easily lead one to believe that the compound or Snake-curve are the best solutions [Jag90a]. The analysis obviously lacks the standard access pattern of DBMSs, i.e., the page (also referred to as block) based access pattern. The results about average runs applies to a database storing one point on a page. Putting a set of consecutive tuples onto a page will result in ignoring discontinuities of the SFC that are within the page, i.e., with a capacity of four points all the jumps on the finest granularity on the Z-curve will be eliminated.

Thus [Jag90a] has run experiments with a two dimensional universe of 256x256 points. The universe was fully occupied and the space was partitioned into blocks of an equal number of points starting at the origin of the SFC. He varied the page size from 1 to 3000 points and performed all possible queries.

For the 512 possible partial match queries there was a reverse linear dependency of fetched blocks to the block size for the recursive SFCs. With a block size of 1, e.g., one point fits on a block, 256 blocks have to be fetched by all SFCs. With a block size of 3000 points we get $256^2/3000 \approx 22$ blocks for the whole universe and $\sqrt{22} \approx 4.7$ blocks for a partial match query in average. Hilbert-curve and Gray-Code-curve perform equally well followed by the Z-curve doing almost as well. As the Compound-curve has huge discontinuities it benefits from the increase of the block size only when a block contains more than one row, e.g., more than 256 points. With respect to the runs the Z-curve had 50% more runs with a block size of 1, but this advantage decreases quickly and with a block size greater than 30 the recursive SFCs are doing equally well. The average number of relevant items (hits) in fetched blocks was tested for a block size of 30 point and grid sizes from 8x8 to 512x512. Independently from the grid size, the number of relevant points per block was 4 to 5 in average for the recursive SFCs while it was getting worse for the Snake-curve with increasing block size. The Hilbert-curve was superior with ≈ 4.5 followed by the Gray-Code-curve with ≈ 4.4 and the Z-curve with 3.8 relevant items per block in average.

In the following he varied the size of a square query from 2x2 to 20x20 and queried a 128x128 universe and a block size of 30 items at all possible positions. The results show that with growing square size the differences between SFCs reduce to less than a few percent w.r. to the number of runs. The average hits per page were increasing with growing query size and, as expected, the Hilbert-curve was doing best with 17 hits ($\approx 56\%$) of the intems on a page were hits before Gray-Code-curve and Z-curve with 15 ($\approx 50\%$) followed by the Snake-curve with 10 ($\approx 30\%$) hits.

Finally, he analyzed the dependency w.r. to different query shapes and growing universe size. In case of growing grid size the recursive curves perform roughly constant while the Snake-curve deteriorates as discontinuities grow. The hits per block grew linear w.r. to the query size. The average size of the query span was essentially the same for all mappings.

3.3.2 [MJF$^+$01]

Analysis of the Clustering Properties of the Hilbert Space-filling Curve

The authors develop closed-form formulas for the number of runs required to answer an arbitrary formed query shape, e.g., polygons and polyhedra. They validated their results with simulations for two and three dimensional universes with a grid size of 32 to 64 units per dimension and five different query shapes. The key result of the paper is that the number of runs depends on the hyper-surface area of the query and not on its hyper-volume. The number of runs is directly related to the number of intersection of the SFC with the surface of the query shape.

According to their formulas in a two dimensional universe the Z-curve requires for a query Q $\frac{1}{3} \cdot \mathit{perimeter}(Q) + \frac{2}{3} \cdot \mathit{sideLength}(Q)$ where $\mathit{sideLength}$ refers to the unfavored dimension of the Z-curve, which is the first dimension used for bit-interleaving where the average distance of two neighbors is higher than for the other dimensions. The Hilbert-curve requires only $\frac{1}{4} \cdot \mathit{perimeter}(Q)$ runs.

3.3.3 [HW02]

On the Quality of Partitions based on Space-Filling Curves

The authors take the block-oriented access into account by analyzing the shape of regions resulting from a partitioning of the Z-curve and Hilbert-curve in a two dimensional universe. By partitioning a SFC into a set of consecutive intervals we also get a region partitioning of the universe. They compare the resulting regions with the optimal partitioning, i.e., the square. As quality coefficient they define $C(p) = \frac{S(p)}{4 \cdot \sqrt{V(p)}}$ where p is a given interval on the SFC, $S(p)$ its surface, and $V(p)$ its volume.

Simulations had been performed for a 1024x1024 grid and all possible partitions of volumes up to 4000 were examined to obtain the maximum, minimum, and average. In worst case the Z-curve has $C^Z = 1.84$ and the Hilbert-curve $C^H = 1.86$, thus the Z-curve performs better in worst case. In average the Z-curve has $C^Z = 1.44$ and the Hilbert-curve $C^H = 1.38$.

3.3.4 [MAK02]

Performance of Multi-Dimensional Space-Filling Curves

This paper focuses on the different segment types of SFCs, i.e., intervals of length one on the SFC. They define the following segment types for two consecutive points on the SFC. We are using our own notation for their definitions to stay conform with our definitions:

For two consecutive points p, q on a SFC corresponding to two points $\vec{p}, \vec{q} \in \Omega$ with $q = p + 1$ and $p = S(\vec{p}), q = S(\vec{q})$ we define the segment $[p, q]$ to be a:

Jump: w.r. to dimension k iff $|p_k - q_k| > 1$

Continuity: w.r. to dimension k iff $|p_k - q_k| = 1$

Reverse: w.r. to dimension k iff $p_k < q_k$

Forward: w.r. to dimension k iff $p_k > q_k$

Still: w.r. to dimension k iff $p_k = q_k$

The authors present closed formulas of these segment types per dimension and in total for the Z-curve, Gray-Code-curve, and Hilbert-curve. In their analysis they are comparing the percentage of these segment types to the overall number of segments $D*(N^D-1)$ with D being the number of dimensions and N the number of grid cells for up to 12 dimensions. Furthermore, they investigated the standard deviation and the distribution of segments w.r. to each dimension individually.

The percentage of jumps is decreasing while the number of dimensions is increasing. With two dimensions the Z-curve has $\approx 6\%$, the Gray-Code-curve $\approx 12\%$ and the Hilbert-curve 0% (Figure 3.13(a)). The Hilbert-curve has no jumps due to its definition. At first glance it might be confusing for the reader, that the Z-curve has fewer jumps than the Gray-Code-curve and why it is converging to 0%. Naturally one would regard every second segment of the Z-curve as a jump, however the authors analyze the segments for each dimension individually and thus the minor jumps on the finest granularity of the Z-curve are ignored since their distance w.r. to each dimension is 1, although their Manhattan distance is 2. With more than 5 dimensions all curves practically have 0% jumps. The Hilbert-curve and Gray-Code-curve share the same properties for all remaining segment types. Except for the *still* segment the percentage is decreasing with increasing dimensionality and it is independent of the grid size, while the Z-curve has 5-10% more segments. The percentage of *still* segments is increasing with the dimensionality converging to 90%.

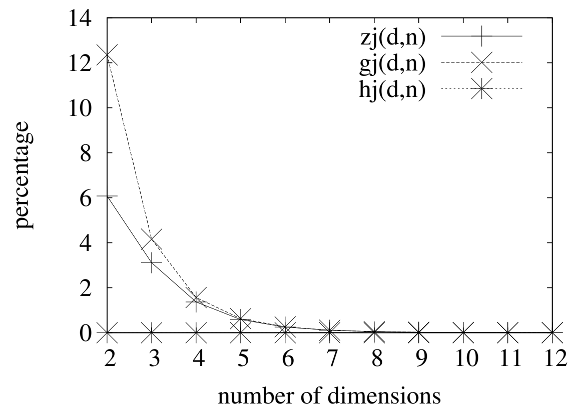
Another interesting result is the relation of segment types w.r. to the different dimensions. The presented results are for a 4d space with grid size 16. The Hilbert-curve has 72% still segments and $\approx 14\%$ forward and reverse segments in every dimension. The Z-curve clearly shows the underlying compound order of the sub quadrants and start with $\approx 7\%$ forward and reverse segments which doubles with every dimensions. With four dimensions they are each at 50% with no still segments left. The Gray-Code-curve has the same properties, but it starts with half the segments for forward and reverse and in average has the same values as the Hilbert-curve while the Z-curve shows a average value which is twice as high.

3.3.5 [Law00]

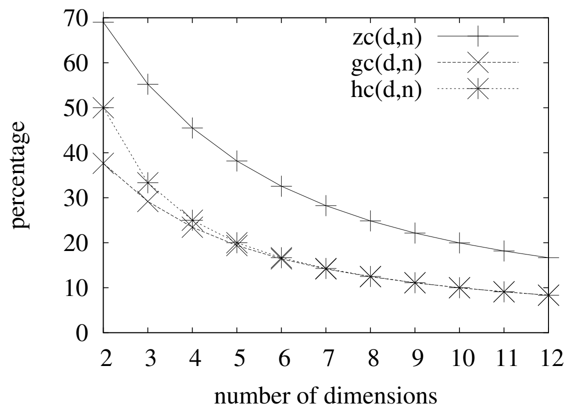
The Application of Space-filling Curves to the Storage and Retrieval of Multi-dimensional Data

Lawder analyses in his thesis all basic curves and operations required for query processing. He provides alternative and optimized algorithms for the calculation of Z-curve, Gray-Code-curve, and Hilbert-curve. For the Hilbert-curve he proposes compressed state-diagrams which work for up to 8 dimensions in order to speed up the calculation of the curve. For higher dimensional universes they are not suitable anymore due to their size requirements.

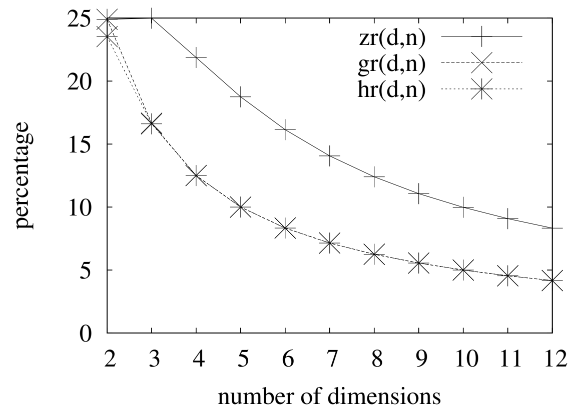
He has also carried out performance measurements with three million randomly generated data points in 3 to 16 dimensional spaces with a grid size of 2^{32} points. The page



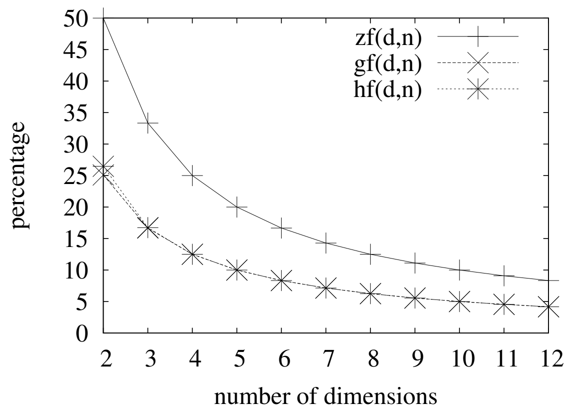
(a) Jumps



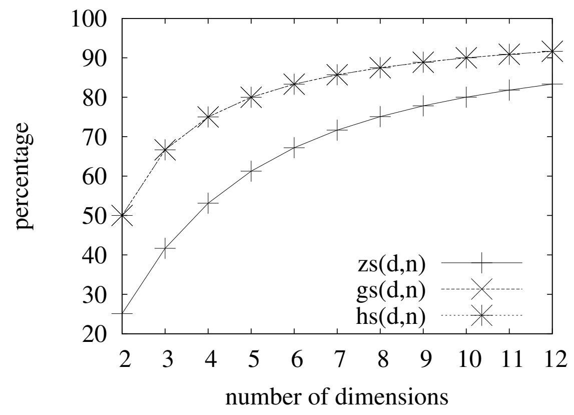
(b) Contiguity



(c) Reverse



(d) Forward



(e) Still

Figure 3.13: Percentage of segments in SFCs

size was adapted to have the same capacity w.r. to the number of records for each dimensionality. In practice there were 8450+/-150 pages which result in $\approx$ 355 tuples per page.

He measured data file creation by random inserts, partial match queries, and range queries for the Hilbert-curve, the Moores Curve (a variation of the Hilbert-curve, [Moo00]), Gray-Code-curve, Z-curve, and Grid-file [NHS84].

The simple design of the Z-curve shows here a considerable advantage over the Hilbert-curve and Gray-Code-curve for both classes of address calculations (S and NJI/NJO) and retains its cost linear to the address length with growing dimensionality. His preliminary measurements also show that the Z-curve is superior w.r. to elapsed time for all measured queries, being faster by a factor of up to 4 for some cases. However it was loading up to 15% more pages in some case and $\approx$ 10% in average for these queries.

3.3.6 Summary of Related Work

First of all our requirements should be mentioned again and then related work is reviewed according to these requirements. In general this thesis considers the management of data in sparsely populated multi-dimensional universes, where we want to:

1. Cluster data spatially,
2. handle arbitrary dimensionality,
3. handle differing dimension cardinalities,
4. efficiently process inserts, deletes, and updates, and
5. efficiently process range queries.

Most related work ([Jag90a, MJF$^+$01, MAK02]) solely analyzes the clustering properties of the different SFCs, i.e., how they preserve the spatial proximity of multi-dimensional points. Specific query patterns, e.g., 2x2 query boxes [Jag90a], partial match queries [Jag90a, FR89], but also generic query shapes [MJF$^+$01] are considered.

However, they only consider completely populated universes resp. the complete mapping of a range query to corresponding set of intervals on the SFC. Raster data (e.g., images, video, audio, etc.) is an example for completely populated universes and the results are valid for these type of data. Typical multi-dimensional applications on RDBMSs (e.g., data warehousing, data mining, spatial data, etc.) are an example for very sparsely populated universes and usually have highly skewed data distributions. The universe is partitioned into regions, corresponding to data pages. The exact partitioning is defined by the one-dimensional ordering of the SFC and the data distribution. Discontinuities of a SFC, like jumps, falling into a region can be neglected now, what matters is the shape of the regions. [HW02] are honoring this in their work and compare the shapes of all possible regions for a given number of segments on the SFC. They compare regions of the Z-curve and

Hilbert-curve with the ideal region shape, the sub-cube. The differences between curves were minimal w.r. to this measure. Considering the compound curve w.r. to this it is clear that it does worse, as a region on the curve might just be a single line in the universe.

While there was a common understanding in the research community, that the Hilbert-curve is the best or among the best possible curves, new insights and qualitative results allowing for a better comparison of the curves have been provided, e.g., [MAK02] showed that differences between curves reduce with increasing dimensionality.

Another lack of existing work is its restriction to cubic universes, i.e., where all dimensions have the same cardinality $|\mathbb{D}_1| = \cdots = |\mathbb{D}_d|$. None of the RDBMS applications we reviewed have this property. For example, in data warehousing there are usually some dimensions with a cardinality (e.g., *product*) which is greater than the other dimensions by several orders of magnitude. When mapping such universes into a cubic universe we get longer keys resulting in a reduced index fanout, i.e., fewer keys fit on one index page, and consequently the index may grow also in height increasing the search and cache cost. Furthermore, address calculations will last substantially longer and when storing the addresses on data pages, they will also reduce the number of tuples fitting to a data page and thus result in more data page accesses for the same result data. For two of our real world data warehouses this would result in an address longer by 50-60% for the indexed dimensions, which had been 3 and 5 out of all defined dimensions.

To the best of our knowledge, [Law00] has been the first considering also the cost of address calculations which are necessary for index maintenance and query processing. As this has been neglected in previous work and one might have thought there is no substantial difference between curves, however there is one, the Hilbert-curve is more costly.

3.4 Comparison of SFC Properties

Different properties of SFCs have impact on the efficiency of a SFC when used for indexing. First, we list metrics considered as important and see how they influence maintenance and query operations. A summary table closes this chapter.

Address Calculation: Address calculation is required for the basic operations of insertion, deletion, and update. Its complexity should be linear to the address length.

Proximity Preservation: Bulk maintenance operations (e.g., loading, deletion and updates) as well as range queries usually process data that is local w.r. to its spatial proximity. The perfect mapping is one where any multi-dimensional range is mapped into exactly one one-dimensional interval and thus access operations are not random, but local. However, this is not possible in general, since the dimensionality is reduced with a SFC mapping.

Jumps: A SFC has jumps if two neighbors on the SFC are not neighbors in Ω. The longer jumps are the more locality is destroyed. Thus an SFC without jumps is better than one with jumps w.r. to this metric. Few jumps have a smaller impact than many jumps.

Average Neighbor SFC-Distance: The average SFC-distance of neighbors in Ω and its standard deviation are an indicator for better proximity.

Symmetry: A SFC is considered to be symmetric if the average SFC-distance of neighbors in one dimensions is equal to the average SFC-distance of neighbors in other dimensions.

Universality: Support of arbitrary dimensionality and dimensions with different cardinalities.

In [Mar99] the symmetry of SFCs (Compound-curve, Z-curve and Hilbert-curve) is discussed according to the average SFC-distance of neighbors w.r. to one dimension. The results clearly show an advantage of the fractal curves over the Compound-curve. Furthermore, the degree of symmetry (the negative standard deviation of the neighbor distances) is better for the Hilbert-curve.

3.4.1 Region Partitioning

This section reviews the region partitioning of SFCs, i.e., what area in Ω corresponds to an interval on the SFC. This is an important property as it has a direct influence of the performance on range queries which are retrieving the set of pages corresponding to the set of regions intersection the query box. Regions consisting of disconnected areas can cause additional intersections and thus deteriorate the query performance.

According to Definition 3.8 on page 31 a *region* is the area covered by all points with addresses included in the SFC-interval. Continuous SFCs like the Hilbert-curve and the Snake-curve create only regions consisting of one contiguous area.

However, due to jumps of the SFC the region corresponding to a SFC-segment is not necessarily one joint area, but it might consist of several disjoint areas. In worst case all of the points in the segment might be apart from each other, but then we would not speak of a proximity preserving SFC anymore. Thus there is an upper bound for the number of disjoint areas a region consists of. Examples of such regions are depicted in Figure 3.14.

A region of the Compound-curve has at most two disjoint areas. The proof comes directly from its definition. Jumps occur just between rows, after making a jump the next jump can only occur, when the row the first jumps was going to has been fully scanned, but if the row is complete it also touches the point from where the first jump starts. Thus disjoint ares can only occur if there is no complete row within the SFC-segment w.r. to the dimension of the major jump.

For the Z-curve [Fri97, Mar99] have proven that there are at most two disjoint areas for an arbitrary Z-region independent from the dimensionality.

Regions of the Gray-Code-curve consist of $2 \cdot (s - 2)$ disjoint areas at most, where s is the number of steps of the curve. The fractal curves discussed in this thesis all use an underlying quad-tree and only differ in the numbering of sub-quadrants. Thus a region

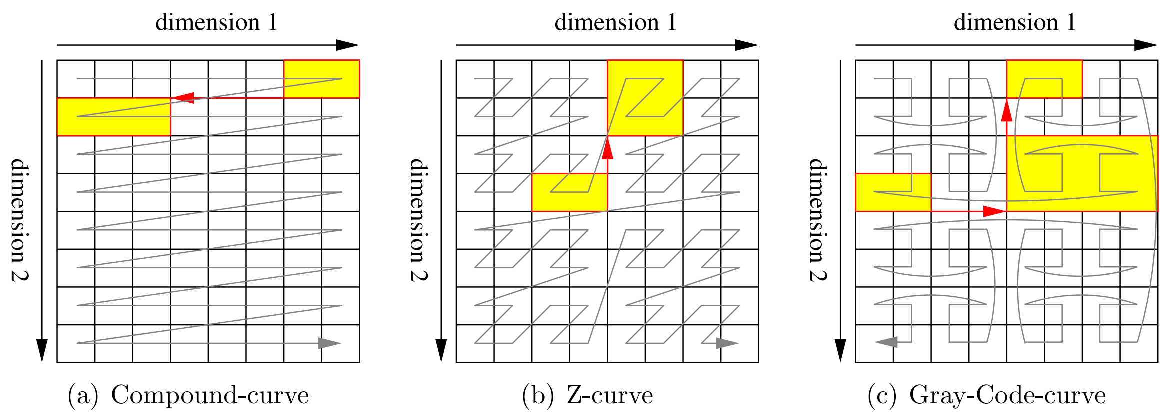


Figure 3.14: SFC-Regions consisting of disjoint areas in Ω

$[\alpha, \beta]$ is the set of quadrants with addresses included in the SFC-segment. Starting with the quadrant on the finest granularity containing α quadrants increase and decrease again when reaching the quadrant on the finest granularity containing β. This corresponds to the traversal of the nodes in the quad-tree from α to β on the leafs of the tree while the common prefix of the two determines how far the traversal is going up to the root node of the tree. The Gray-Code-curve has three jumps on the second level of recursion (Figure 3.8(b)) for the two dimensional case and this basic pattern continues on finer levels. Disjoint regions can only occur due to a jump from one quadrant to the next one. When a quadrant or half of a quadrant has been traversed the jumps within these cannot cause any disjoint regions, since all points within them have been traversed. Thus a region consists of a sequence of increasing an then decreasing quadrants and there is just one jump that can cause a disjoint region, i.e., a jump from a smaller quadrant to a bigger one or the reverse. With s steps there a just $s - 2$ jumps between those quadrants possible. Therefore, the number of jumps of the Gray-Code-curve is dependent on the size of the region.

3.4.2 Summary

Finally Table 3.3 gives a summary of SFC properties and complexities for the SFCs considered in this chapter. Note that the Hilbert-curve and Gray-Code-curve have substantially higher complexities for the $NJI()$ and $NJO()$ calculations as there are, to the best of our knowledge, no better algorithms available. The complexity of $S()$ and $S^{-1}()$ for the Hilbert-curve are from [Law00] page 89. While the Snake-curve is the best choice w.r.t. to these metrics it does not preserve proximity as well as the fractal curves.

When considering Z-curve vs. Hilbert-curve for an application, it is important to be aware of the differences:

I/O: The Hilbert-curve has a higher locality resulting in fewer page accesses. The actual advantage depends on the typical query shape and size of the pages/tuples,

SFC	Jumps	# Region Areas	$S()$	$S^{-1}()$	$NJI()$	$NJO()$	$\lvert\mathbb{D}_i\rvert \neq \lvert\mathbb{D}_j\rvert$
Compound	+	≤ 2	$O(d)$	$O(d)$	$O(d)$	$O(d)$	+
Snake	-	1	$O(d)$	$O(d)$	$O(d)$	$O(d)$	+
Z	+	≤ 2	$O(a)$	$O(a)$	$O(a)$	$O(a)$	+
Hilbert	-	1	$O(2a)$	$O(2a)$	$O(tt)$	$O(tt)$	-
Gray	+	$\leq 2(s-2)$	$O(a)$	$O(a)$	$O(tt)$	$O(tt)$	+
Bit-Perms	+	≤ 2	$O(a)$	$O(a)$	$O(a)$	$O(a)$	+

Where d is the dimensionality, a the address length, s the number of steps, $tt = O(\frac{a^2}{d \cdot 2^d})$ the complexity for a tree-traversal, and $i, j \in D \wedge i \neq j$.

Table 3.3: Tabular overview of SFC properties

i.e., the advantage of the Hilbert-curve are weakened the more tuples fit on a page. Performance advantages range from $\approx 30\% to 0\%$.

CPU: Algorithms for address calculation, NJO, and NJO are substantially more expensive for the Hilbert-curve and may deteriorate the advantages of its better clustering. Not only the address length, but also the dimensionality influences the complexity. To the best of our knowledge, there are no algorithms without these deficits for the Hilbert-curve and using pre-built state-diagrams is not a flexible alternative.

Structure: The Hilbert-curve is designed for universes where the dimension cardinalities are equal or only differ slightly. Indexing universes with highly differing dimension cardinalities leads to longer addresses, and when data is embedded wrongly, to bad clustering (see Figure 3.7).

The question is:

“**Which SFC should I use for my application?**”

and the answer to this question is:

“**It depends!**”

Summarizing our result we give the following advice resp. some rules of thumb:

- For most applications the Z-curve is adequate, i.e., more efficient or efficient enough!
- If address calculations are frequent and are likely to become a bottle neck, then the Z-curve is the right choice as it is more efficient and easier to implement.
- If your dimensions do not have the same cardinality, use the Z-curve, its addresses will be shorter and it clusters as expected, i.e., no dimension is significantly preferred.

- If optimal clustering is the only thing that matters, the Hilbert-curve is the right choice for you, but ensure you are embedding universes in the right way unless all dimensions have the same cardinality.
- Make sure to review the access pattern, i.e., the query shapes, page size, etc., maybe the Gray-Code-curve or some other clustering is suited even better!
- If you run range queries, ensure you are clustering data, any clustering is better than no clustering!

.

Chapter 4

The UB-Tree

Figure 4.1 on the next page gives a first glimpse to the UB-Tree. The original two dimensional UB-Tree has been projected to a globe. The underlying data are tuples $(longitude, latitude)$ where all points have the same height difference to their neighbors. The actual data has been generated from a map image. A region with the same color is a region of the UB-Tree. The denser an area is populated, the smaller regions get, e.g., the sea consists of few big regions and mountains of many small regions. Two important properties of the UB-Tree are already visible, it partitions the universe into disjoint regions and adapts to the data distribution.

In the following, this section introduces the basic concepts of the UB-Tree, a descendant of the B^+-Tree, which is designed to efficiently support range queries on multidimensional point data. Building on the concepts introduced in Chapter 3 we discuss the properties which are specific to the UB-Tree and required in the following chapters.

An overview on the different implementations of the UB-Tree is given. This is important, as certain effects occurring in measurements are dependent on the implementation used for a measurement.

Further, we introduce techniques for efficient bulk insertions and deletions. The presented algorithms are not restricted to the Z-curve, but they work independently from the applied SFC.

4.1 Basic Concepts of UB-Trees

The UB-Tree [Bay97, Mar99] is a clustering index for multidimensional point data, which inherits all the properties of the B^+-Tree [BM72]. Logarithmic performance guarantees are given for the basic operations of insertion, deletion, and point query and a page utilization of 50% is guaranteed. It utilizes the Z-curve to map the multidimensional space to a one-dimensional space, partitions it and indexes it with a B^+-Tree.

Its sophisticated algorithms for multidimensional range queries [Mar99, Ram02] offer excellent properties for multidimensional applications like DWH, GIS, archiving systems, temporal data management, etc. We have shown that integrating the UB-Tree into a

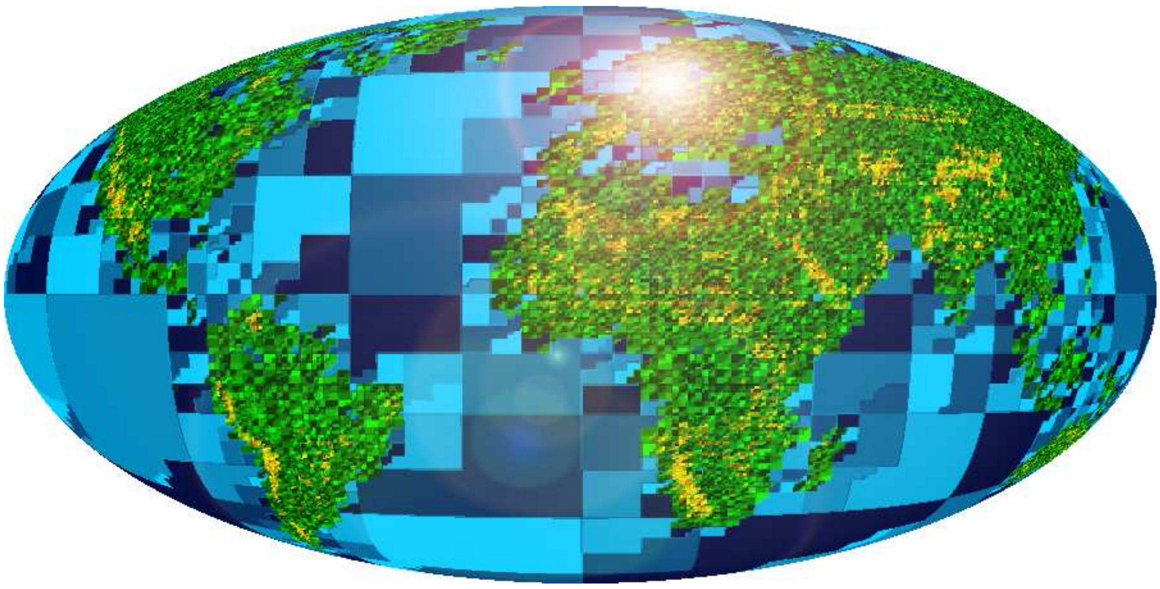


Figure 4.1: A UB-Tree Partitioning the World into Z-regions

RDBMS providing a B$^+$-Tree can be done with reasonably small effort [RMF+00, Ram02], since the UB-Tree is the multidimensional extension of the B$^+$-Tree. The integration offers the advantages already discussed in Section 3.2.3.

The Tetris algorithm [MZB99, Zir03] for sorted reading of multidimensional ranges allows efficient pipe-lining and avoids external sorting thus delivering the first results instantly and being faster in the overall performance. Building a UB-Tree from a pre-sorted input is handled by the Temptris algorithm [Zir03]. Both algorithms utilize a sweep line to separate data into processed/stable and to-be-processed data. Combining them enables the efficient calculation of aggregation-networks required by DWHs.

Efficient processing of data organized in hierarchies is discussed in [MRB99, Pie03]. [Pie03] describes a generic model, its application to the UB-Tree, the integration of the technique into Transbase and performance evaluations.

Thus, the term *UB-Tree* refers not only to the mapping and partitioning of a multidimensional universe, but also to the advanced query processing algorithms as described above. The combination of these allow for an efficient management and query processing of multidimensional point data.

4.1.1 Z-regions

The UB-Tree introduces the new idea of partitioning the data space into disjoint but adjacent *Z-intervals* , which are mapped to data pages. The Z-intervals are indexed by a B$^+$-Tree. This allows for a region based access to the data, which is also the major difference compared to former Z-curve-based indexing techniques, i.e., [OM84, Ore90].

The Z-intervals partition the universe completely and without overlap. Due to this it is sufficient to index only the ending resp. the starting Z-address of Z-intervals, since the starting address of a region can be calculated by incrementing the ending address of the previous region. In the following we assume that we are indexing the last address within a region. A Z-interval represents a subspace of the universe and we will refer to this space as a *Z-region* (see Definition 3.8 on page 31). Furthermore, as the two terms are equivalent we will use the more intuitive one where appropriate.

Page overflows during insertion are handled by splitting the affected page in the middle w.r. to the tuples, i.e., we calculate a *splitting Z-address* σ partitioning the tuples on the page into two equal sets w.r. to their Z-addresses. The first half of the tuples from the old page are moved to a newly created page corresponding to the Z-interval indexed by the separating Z-address σ. The old page is updated, the new page stored and the new split address referring to the new page is inserted into the B$^+$-Tree. An underflow during deletion is handled by merging two adjacent regions, i.e., by the default algorithm of the B$^+$-Tree.

Formally, the structure of the UB-Tree can be defined as a set of k Z-intervals as follows:

Definition 4.1 (UB-Tree-Structure)

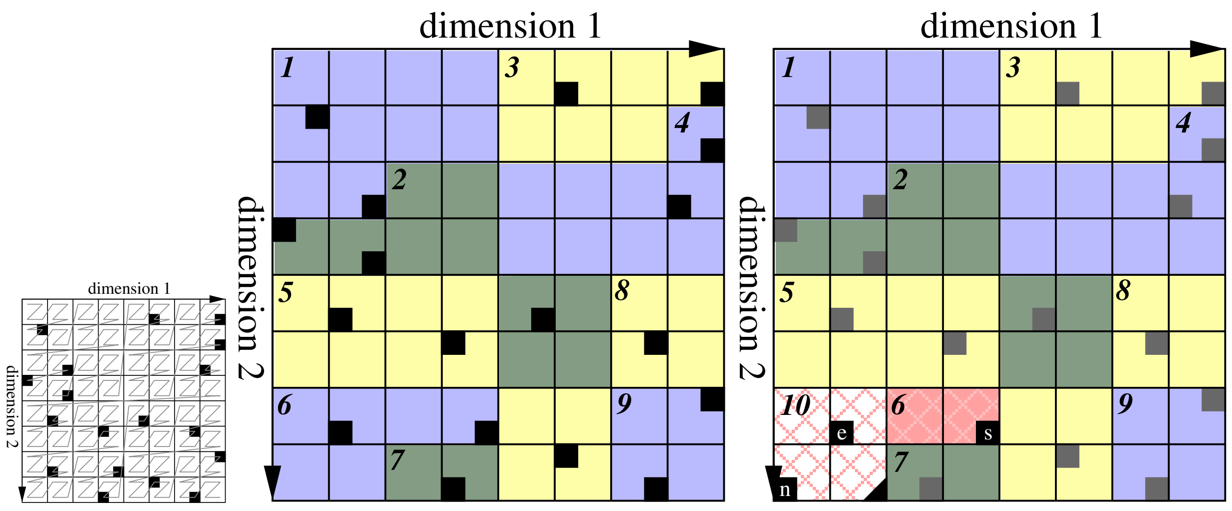
A UB-Tree is the partitioning of a multidimensional universe Ω into a set of intervals $\mathbb{U}(\Omega) = \{[\sigma_1, \epsilon_1], [\sigma_2, \epsilon_2], \ldots, [\sigma_k, \epsilon_k]\}$ on the Z-curve where $\sigma_{i+1} = \epsilon_i + 1 \; \forall i \in [1, k-1]$. The intervals are indexed by a B$^+$-Tree indexing in the interval end and correspond to data pages. $\diamond$

This is a B$^+$-Tree, where the key of a point is calculated by the Z-function $Z(\vec{p})$. Furthermore, the split address (the separators stored in the index part) are chosen to take the region shape into account, i.e., to create regions with as few fringes as possible.

The basic operations of insertion, deletion, and update of a tuple $\vec{p}$ are handled by calculation $Z(\vec{p})$ and doing a B$^+$-Tree search to find the data page containing $\vec{p}$, i.e., $Z(\vec{p}) \in [\sigma_i, \epsilon_i]$. Storing the tuples on data pages in address-order allows for efficient binary-search and split while adding some additional cost for maintaining the order during insertion and update.

Example 4.1 (Insertion into a UB-Tree)

Insertion of a new tuple at position $(0, 15)$ (at bottom left corner labeled n) into the two dimensional UB-Tree on a $16x16$ universe as depicted in Figure 4.2(a) is performed as follows. We calculate the Z-address of this point and then locate the Z-region containing this point, which is region *6* (Figure 4.2(b)). Now we retrieve the page corresponding to this region and insert the point. With a page capacity of two points, a split is necessary, creating region 10. The separator between point e and s is chosen to create two regions with as few fringes as possible, i.e., a square for region 10 and a rectangle for region 6. Finally, the two pages are stored and a new separator for region 10 is inserted (Figure 4.2(c)). $\diamond$



(a) data points (b) Z-region (c) Z-region after split

Figure 4.2: Z-regions of a UB-Tree for a given Data Distribution

4.1.2 Range Query Processing

In order to query a UB-Tree we need additional algorithms utilizing its structure.

Range query processing is performed as described in Section 3.2.3. The core of the processing is its efficient NJI-algorithm developed by [Mar99] and in detail described and analyzed by [Ram02]. Additionally, [Ram02] provides a NJO-algorithm to avoid post-filtering of Z-regions lying completely within the query box. Their complexity is linear to the address length enabling the efficient processing of range queries independently on the number of dimensions.

An algorithm handling a set of range queries was developed by the author during his master thesis [FMB99]. Whenever an attribute is restricted to more than one interval this results in a set of query boxes. Processing them sequentially can cause multiple page accesses to the same pages and accesses not in Z-order. The presented algorithm avoids this by processing the query box set simultaneously.

4.2 UB-Tree Implementations

Several implementations of the UB-Tree have been made during the MISTRAL-project. Their differences are presented in this section. As most measurements have only been conducted with one specific implementation, it is important to be aware of effects caused by it.

attribute name	data type	description
address	bit-string	upper bound of the Z-region
count	integer	number of tuples stored on the UB-page
region	varchar	content of a Z-region

Table 4.1: UBAPI: DBMS-relation storing a UB-Tree

4.2.1 UBAPI

The *UBAPI* (**UB**-Tree **ap**plication **i**nterface) is the first implementation of the UB-Tree. It is a middle-ware between the application and a DBMS implemented as an ANSI C application-library. It re-implements the basic data types, i.e., attribute, tuple, relation, and page. A page of the UBAPI is represented by a tuple (Table 4.1) indexed by a B^+-Tree. Thus the DBMS is used only as a storage manager supporting B^+-Tree-lookups. Own meta-data tables store the structural information about UB-Trees. The relation storing a UB-Tree is shown in Table 4.1 with a clustering B^+-Tree index on the Z-address. The data-types of the attributes differ slightly depending on the types available by the used DBMS.

The UBAPI was easily created on top of an DBMS allowing a proof of concept, but it has several drawbacks:

- standard techniques of the DBMS are not reused, i.e., data structures and functions for attribute, tuple, relations, page, etc. are re-implemented.
- it has no standard interface and it is only partially complete, i.e., some functions are not fully implemented; one has to use special tools for creation, loading, and querying.
- there is no query language and no optimizer.
- the interprocess communication between client and DBMS is increased. As query processing is done within the UBAPI-library on the client side, not only the result tuples are transferred to the client but all candidates, i.e., whole pages.

Nils Frielinghaus and Volker Markl have created the initial version of the library on top of Transbase. Various students have ported it to other DBMSs and extended its functionality. The following RDBMS are supported: Transbase, DB2, Informix, Oracle, and Microsoft SQL-Server.

Its superiority for multidimensional problems in comparison to the standard indexes, i.e., secondary indexes, compound indexes, bitmap-indexes, provided by these systems has been shown in various technical reports and master theses [Fri97, Pie98, Bau98b, Mer99] and publications [Bau98a, FMB99, MZB99, Mar99, MRB99].

4.2.2 Transbase ® Hypercube

In an EU-funded project the UB-Tree technology, i.e., the basic algorithms for Z-address calculation, split address calculation, and range query handling, have been integrated into the kernel of the RDBMS Transbase [TAS00, RMF$^+$00]. Furthermore, extensions of the data definition language (DDL) and the optimizer have been implemented in order to enable the UB-Tree for generic query-processing.

The new version of Transbase featuring the UB-Tree is called "Transbase Hypercube" and is available as a commercial product. It is already successfully running at several companies.

In comparison to the UBAPI, using a UB-Tree as index now requires only modifications of the DDL statement creating the index, no other application code is affected. Thus the UB-Tree can be used transparently like any other index provided by the DBMS. Furthermore, the implementation requires, fewer LOCs (lines of code), i.e., only the core components making up a UB-Tree have have been implemented. Also the performance improves due to reduced interprocess communication, i.e., only result tuples are transfered to the application.

However, the code base of Transbase is rather complex as it is a commercial product addressing all aspects of a relational DBMS and it is not freely available for research due to commercial interests.

4.2.3 RFDBMS

Further research on new indexing methods, e.g., variants of the UB-Tree, and optimizations were not possible with the previous implementations due to the following reasons:

UBAPI: the UBAPI was not offering access to the implementation of methods of the underlying DBMS, i.e., the storage manager, optimizer, and the B$^+$-Tree.

Transbase Hypercube: Transbase Code is protected due to commercial interests and is not available for research.

Before considering an implementation from scratch we inspected the following existing projects as an alternative starting point for the new UB-Tree implementation: **Commercial DBMSs:** Function-Based B-Trees, interfaces to extend indexes and **free DBMSs and code:** GiST, Shore, Predator, PostgreSQL, java.XXL. However, all of them have severe drawbacks and thus we decided to start from scratch and built a new implementation satisfying our requirements while remaining small and easy to maintain. In the following we discuss the drawbacks of the existing code bases and why they were no option for us.

Function-Based B-Trees are using a function F to compute the key for tuples. Commercial implementations are: Oracle8i [Ora99], IBM DB2 [CCF$^+$99], and MS SQL Server 2000 (computed columns [Mic00]). However, they do not allow for integrating of new split and query algorithms (e.g., UB-Tree range query). Therefore, implementing the UB-Tree with $F =$ Z-value will not lead to the expected performance.

Interfaces to Extent Indexes: Some commercial DBMSs provide enhanced indexing interfaces allowing for arbitrary index structures in external modules: Extensible Indexing API [Ora99] and Datablade API [Inf99]. The user has to provide a set of functions that are used by the database server to access the index either stored inside the database (i.e., mapped to a relation in Oracle) or in external files. However, neglecting the cost of ownership for a commercial DBMS, only non-clustering indexes are supported inside the DBMS and internal kernel code cannot be reused for external files, leading to a significant coding effort for re-implementing required primitives.

The General Search Tree (GiST) approach: GiST [HNP95] provides a single framework for any tree-based index structure with the basic functionality for trees, e.g., insertion, deletion, splitting, search, etc. The individual semantics of the index are provided by the user with a class only implementing some basic functions. The UB-Tree fits into GiST, but an efficient implementation requires more user control for the search algorithm and page splitting as also suggested by [Aok98]. Still, the code does not provide a generic buffer manager and appropriate tools for managing, inspecting, and querying data.

Other Options: Shore [Tea00] offers the buffer management we were missing in GiST, but it consists of too much source code. For similar reasons we voted against an integration in PostgreSQL [Tea03a], Predator [Tea03b] or XXL [jT03] as these systems were providing functionality (e.g., locking, SQL interface, Client/Server architecture) irrelevant for our goal of comparing indexes.

Due to these reasons, a completely new implementation was founded by the author and a master student, Oliver Nickel. It was called RFDBMS (**r**esearch **f**ocused **DBMS**) and it was presented as a software demo at EDBT'04 [Wid04].

It reassembles a simplified DBMS in the C++ programming language. Included is a storage manager, basic data structures (Attribute, Relation, Tuple) and various indexes (B^+-Tree, R^*-Tree, UB-Tree, BUB-Tree, etc.). Thus it was possible to modify and optimize all aspects of query processing. Due to its component based approach it was also possible to easily extend existing indexes and add new indexing techniques. As the same components are used by all indexes, their comparison is fair as it only shows the difference between indexes and not independent implementations. Thus it eliminates the overhead of the UBAPI implementation, i.e., reduced page capacity due to additional management data, additional interprocess communication and re-implementation of concepts like page and tuple causing additional CPU overhead.

Table 4.2 shows the number of LOCs (Line of Code) and the inheritance graph for some of the implemented indexes. It shows that the UB-Tree inherits most code from the B^+-Tree. In comparisons with the R^*-Tree this requires substantially fewer and easier code and in fact the sum of the number of LOCs for the B^+-Tree and UB-Tree are approximately the same as for the R^*-Tree.

4.2.4 Summary

All three implementations have their advantages and drawbacks. The UBAPI has been a proof of concept, while the integration of the UB-Tree in Transbase has enabled commercial

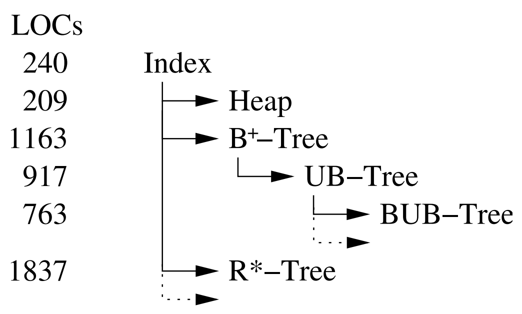


Table 4.2: Comparison of Index Implementations in RFDBMS

System	**LOCs**	**Pros & Cons**
UBAPI	≈ 85000	+ proof of UB-Tree concept for all major RDBMSs − re-implementation of DBMS primitives − no standard interface and query language − incomplete − implementation overhead: interprocess communication, tuple processing − no access to DBMS specific code, e.g., DBMS primitives, storage manager, B^+-Tree
Transbase HyperCube (only UB-Tree specific code)	≈ 10000	+ full featured and stable commercial product with support + “no modifications” of application code except DDL statements + implements only UB-Tree relevant code, all other primitives and techniques are reused + optimal performance due to tuple and predicate processing within in the kernel − source is protected by commercial laws and rather complex
RFDBMS	≈ 20000	+ full access to source code including DBMS primitives and storage manager as well as index implementations + adding new indexes is easy + limited to the essential code necessary for a meaningful index comparison − only for research purposes, i.e., it is only partially complete and not suitable for commercial applications

Table 4.3: Summary of UB-Tree Implementations

applications. RFDBMS has enabled further research on UB-Trees. Table 4.3 shows a summary of the implementations, listing their average number of LOCs, and the advantages and disadvantages of each.

4.3 Bulk Insertion

This section considers the issue of bulk loading large data sets into a table organized as UB-Tree. Especially for data warehousing, data mining, and OLAP it is necessary to have efficient bulk loading techniques, because loading does not occur continuously, but periodically with usually large data sets.

Two techniques are proposed, one for initial loading, which creates a new UB-Tree, and one for incremental loading, which adds data to an existing UB-Tree. Both techniques try to minimize I/O and CPU cost. Measurements with artificial data and data of a commercial data warehouse (DWH) demonstrate that our algorithms are efficient and able to handle large data sets. As well as the UB-Tree, they are easily integrated into a RDBMS.

4.3.1 Introduction

In case of loading a huge amount of data into an indexed table, it is usually not feasible to use the standard insert operation of the index, since this would result in a big overhead caused by unnecessary page accesses. In most cases, tuples emerge in random order or an order, which is not suitable for the used index and therefore will lead to random inserts. With *random insertion* we denote that consecutive tuples will be inserted into different pages and consequently an index search and a data page access is necessary for nearly each insertion. In the worst case, the cost for loading will not increase linear to the number of pages required to store the tuples but linear to the number of new tuples.

Additionally, the query performance is of crucial interest and it depends on the clustering and page utilization.

So far there has been already some work on bulk loading of multidimensional index structures. Various papers exist on this subject for other multidimensional index structures, e.g., for R-Trees [RL85, KF93, LR95, LEL97, GLL98], Grid-files [LN97], and quad-trees [HSS97]. [Gra03a, Gra03b, Gra04] addresses B$^+$-Tree bulk operations by partitioning the data by an artificial leading column defining the partitioning. Thus bulk insertion is handled efficiently and concurrently to queries, as it only affects a new partition, but not the old data. Loading the partition is handled like initial bulk loading.

In this section we present bulk a loading algorithm for the UB-Tree, allowing to efficiently perform bulk insertion for initial and incremental loading.

[Bau98b] has implemented a simple bulk loading tool for initial loading within the Mistral-Project.

Our contribution is the discussion and description of two simple and efficient bulk loading algorithms for the UB-Tree, one for initial bulk loading and another one for incremental bulk loading. The reuse of existing techniques is discussed, which have been tested and proven to be robust and fast in practice. This simplifies the integration of these algorithms to a large extent. Parts of this work have been published as [FKM$^+$00] and a Japanese patent has been filed on 22 May 2000 for the incremental bulk loading technique (Application number: 2000-149648).

4.3.2 General Problem Description

Efficient bulk loading is crucial when loading a huge amount of data into an indexed table. Creating a new index should happen as fast as possible providing the best possible performance for queries. According to [Pal00] loading data into a data warehouse is one critical process in the initiation and maintenance of a data warehouse application. The sheer volume of data involved dictates the need for a high-performance data loader. The ability to load vast data sets efficiently reduces the overall cost associated with the maintenance of a DWH.

The source data is usually provided in some kind of portable format, e.g., ASCII flat files, XML, Excel-Table, etc., but not in a binary format. This happens especially for data warehousing and data mining applications when moving data from the OLTP systems to the OLAP system.

The main goals, which should be achieved by bulk loading techniques, are the following ones:

1. Minimize random disk accesses,
2. minimize disk I/O,
3. minimize CPU load,
4. optimize clustering and
5. optimize page utilization.

Random insertion cannot achieve these goals: each single insertion may trigger a random disk access causing I/O as subsequent insertions must not be local w.r. to their position in the universe (goal 1 & 2); each single insertion causes a tree traversal causing CPU-load (goal 3); page clustering depends on the order of page creations caused by the insertion order (goal 4); only the standard page utilization of 50% resp. 66% is guaranteed.

The main cost for loading arises by accesses to secondary storage, strictly speaking the loading process is I/O bound. Therefore, it is essential to minimize disk I/O and especially to avoid random accesses wherever possible. On the other hand linear disk accesses are quite fast due to caching techniques of todays hard disks and operating systems. Consequently, the CPU load is also to be recognized as critical factor for the loading process and should be minimized as well.

With respect to query performance it is important to provide good clustering, since this reduces random disk accesses. There are two types of clustering, namely tuple and

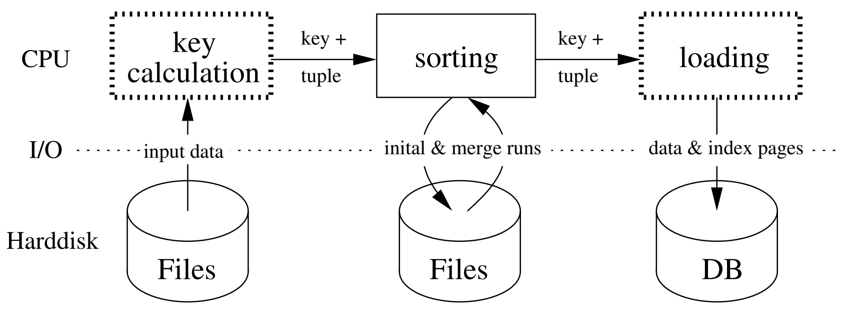


Figure 4.3: A Bulk Loading Architecture for one dimensional clustering Indexes

page clustering (see Section 2.4 on page 21). For bulk loading it is possible to achieve both, in other words not only the tuples within one page are clustered, but also the pages are clustered according to the used index.

Additionally, a high page utilization should be achieved in order to decrease the number of pages to load for answering queries. For databases, which are only loaded once without ever adding new data to existing pages, e.g., CD-ROM databases, the pages should be filled up to their maximum. Databases with new data also going to existing pages will not be created with a 100% page utilization, as this would cause a lot of splits at the beginning of loading new data. Using a page utilization below 100% avoids this. A given page utilization can be guaranteed for initial loading, but for incremental loading only 50% is guaranteed resp. 66% with the so called overflow technique.

4.3.3 Algorithms

Multidimensional bulk loading algorithms can be classified according to three groups:

1. algorithms which apply a certain partition to the multidimensional input data and load those partitions into the index,

2. algorithms which apply a total sort to the input data and load the sorted data into the index and

3. algorithms which utilize pre-sorted data to avoid a total sorting ([Zir03]).

In the first category algorithms like [GLL98] for R–Trees and [LN97] for Grid–Files are available, while the second group applies to [KF93] for R–Trees which sorts the data according to the Hilbert curve as well as to the algorithms we present here.

Only the second group can guarantee the best possible clustering and desired page utilization.

The bulk loading algorithms in the second category have three processing steps in common (Figure 4.3), first they calculate the key for each tuple of the data set, second the tuples are sorted according to the key and third the sorted data is loaded into the index.

In order to speed up the whole process it is useful to arrange these steps in a pipeline as depicted in Figure 4.3, since this avoids writing temporary results between processing steps to secondary storage, i.e., between the calculation and the sort step and between the sort and the loading step. Additional performance can be gained by using a binary format for the intermediate results, because it avoids conversion from internal binary representation and external ASCII representation and vice versa. This saves usually some disk I/O, but, what is even more important, saves a lot of CPU time, because subsequent parsing and key calculation for each tuple is avoided. The sorting step is a pipe-line breaker, i.e., loading begins when the complete input data has been sorted and the final merge starts.

4.3.3.1 Key Computation

The key used to identify a tuple is usually a subset of the attributes of a tuple. However, internally the used index might use a different representation. For example compound B$^+$-Trees use a concatenation of the key attributes, but with space filling curves there is an additional computation, which calculates the scalar value on the curve representing the point specified by the key attributes resp. coordinates. B$^+$-Trees based on such a computation are called functional B$^+$-Trees. The UB-Tree uses such a B$^+$-Tree and adds calculation of the Z-address. This calculation is efficient, using bit interleaving of the key attributes (the dimensions) [OM84].

The keys are also used in the sorting and loading process and therefore it is more efficient to add the key to each tuple of the temporary data sets in order to reuse it for the subsequent processing steps than to recalculate it each time. This and using a binary format for the intermediate files saves CPU load.

4.3.3.2 Merge Sorting

The best choice for external sorting of large data sets, that do not fit into main memory, is the merge sort algorithm [Knu73]. It pre-sorts junks of the data set, which fit into RAM and writes them back to disk as initial runs. The best strategy to get the longest possible runs is to use a heap for internal sorting, i.e., one gets $2M$ long initial runs [Knu73], where M is the number of tuples which fit into internal memory. Those initial runs are then merged, where as much files as possible are merged at once in order to avoid subsequent merge runs requiring additional disk accesses.

The disk accesses necessary for this are sequential accesses, which are quite fast, because of the pre-fetching of hard disks and operating systems. In order to avoid unnecessary I/O resp. disk accesses it is possible to integrate the creation of initial runs into the key computation and perform merging in the second step.

4.3.3.3 Initial Loading

When creating a new index from scratch we speak of *initial loading*. Compared to other multidimensional indexes, loading the UB-Tree is simple, because B$^+$-Tree standard techniques can be used once the data set is sorted according to the Z-address.

```
void SFCIndex::writeLastPages(LargePage &page)
{
  if (page.getUtilization(page) > 100%) {
    Page pg1, pg2;
    splitPage(page, &pg1, &pg2);
    storePage(pg1);
    storePage(pg2);
  }
  else {
    storePage(page); // implicitly convert large page to normal one
  }
}
```

Table 4.4: Writing the last pages after Bulk Loading

For the algorithms we use a page called *large page* which is can store twice the tuples of a standard page. It is used only internally and may have a page utilization of 200% w.r. to a standard page. This allows to simplify the algorithms. All percentage statements, given in the following, are w.r. to standard pages. We also do not discuss the maintenance of the index pages of the underlying B^+-Tree, because this can be achieved by standard techniques [RS81], i.e., by caching the index pages, leading to the current data page.

The initial loading algorithm works as following: It starts with an empty large page called the `page`. Now it reads tuples and adds them to this large page until it is full resp. reaches twice the desired page utilization. When it reaches this limit, it splits the current large page in the middle into two normal pages `pg1` and `pg2` containing each one half of the tuples of the large page. `pg1` is now complete, it has the specified page utilization, and is written to disk. The new separating address is inserted into the index part. This does not require any search in the index, as we are only appending to its end, but it may trigger splits of index pages by page overflows. Using a large page for the index pages in the path leading to the current data page can guarantee the desired page utilization also for the index pages.

The contents of `pg2` is copied to the large `page`. The algorithm continues now adding tuples to the large page and splitting it as before, until no more tuples are left.

When finished, it has to check, if the large `page` needs a final split before storing it, since it might be filled to more than 100% of a standard page. If so it splits the page and stores the two new standard pages `pg1` and `pg2` otherwise the page can immediately be stored as standard page, because its utilization is less or equal than 100%. The function for storing this last large `page` is shown in Table 4.4.

This algorithm allows for a guaranteed page utilization. An exception are the last two pages which might be filled only by 50%. The algorithm provides tuple clustering as well as page clustering, since the input data set is already sorted according to the Z-curve. In case of CD-ROM databases one can choose the maximum page utilization, which is 100%.

This algorithm can also be used to reorganize an existing UB-Tree, by reading the tuples not from a flat file, but from an existing UB-Tree. Adding an existing UB-Tree to the merge phase allows for the efficient creation of a new UB-Tree containing both, new and old data. This method (*initial merge loading*) is superior when new tuples are added to each page of the existing UB-Tree. It guarantees a specified page utilization and optimal clustering. However, when new tuples contribute only to a small subset of the existing UB-Tree-pages by storing the majority of tuples on new pages, it is faster to use an incremental loading algorithm – although it is no longer possible to hold the strict guarantees for clustering and page utilization.

Usually, it is known in advance what loading technique fits best to the actual application or data set and thus allows for choosing the right one.

4.3.3.4 Incremental Loading

When extending an existing UB-Tree we speak of *incremental loading*. Incremental loading differs from initial loading, because it does not only create new pages, but additionally updates existing pages.

The algorithm works as follows: We insert the first tuple into the UB-Tree and set the large `page` to the page the tuple was inserted to. Now the main loop of the algorithm starts.

The next tuple is read from the input data and checked if it tuple belongs to the current `page`. If not, the current large `page` is stored with the function `writeLastPage`, which cares for a split. Now the page to which the tuple belongs is retrieved from the UB-Tree. This page can be found by a point search for this tuple. The page content is copied to the large `page` which is used internally. The tuple is inserted into the large `page`. If the large `page` is full w.r. to the designed page utilization, it is split into two normal pages `pg1` and `pg2`. The page containing the inserted tuple becomes the large `page` and the other one is written to disk. The algorithm continues adding tuples to the current `page`, checking and splitting it as before until no more input tuples are left. Finally, the large `page` is stored with the function `writeLastPage`.

Table 4.5 depicts this algorithm in pseudo C++-code. It should be mentioned that the `storePage` functions needs to check if the page to be stored does already exist or if a new page needs to be created. This can be achieved by marking those pages, which have been retrieved from the UB-Tree. When splitting a page it is necessary to create the new page `pg1` identified by the new split address, but `pg2` only has to be updated, because it retains its Z-address.

Compared to the initial loading algorithm there are two new parts. One is the check if a tuple belongs to the current page and the other, a new strategy for keeping the right page after a page split. The desired page utilization can be specified by `pageUtilization` and will be guaranteed except for the last two pages or the last two pages of a continuous loaded sub-part of the Z-curve occurring when the last page needs to be split before storing. Those pages might be filled only to 50%. The last two pages of a sub-part of the Z-curve occur when a tuple does not belong to the current page (but to another existing page of

```
void SFCIndex::incrementalBulkLoad(TupleKeyStream &tuples,
                                  float pageUtilization = 100%)
{
  Tuple tuple;
  Address address;
  LargePage page = INVALIDPAGE;

  while (!tuples.eof()) {
    tuples >> tuple >> address;
    if (address > page.address || page == INVALIDPAGE)
    {
      if (page != INVALIDPAGE)
      {
        writeLastPages(page);
      }
      page = fetchPage(address);
    }
    page.insertTuple(address, tuple);
    if (page.utilization() > 2 * pageUtilization)
    {
      Page pg1, pg2;
      splitPage(page, &pg1, &pg2);
      if (address <= pg1.address)
      {
        storePage(pg2);
        page = pg1;
      }
      else
      {
        storePage(pg1);
        page = pg2;
      }
    }
  }

  writeLastPages(page);
}
```

Table 4.5: Algorithm for Incremental Bulk Loading

the UB-Tree) and if the current page needs a split before storing. But, in order to increase the page utilization in this case, it is possible to apply the enhancements known for the B*-Tree [Küs83, BY89] i.e., to distribute tuples between consecutive pages in order to gain a higher page utilization.

Page clustering can be guaranteed only for the newly created pages. When the majority of tuples from a new data set contribute to new pages then it is much faster to use incremental loading instead of initial merge loading. Only a small fraction of the existing pages is accessed instead of merging the new data set and an whole UB-Tree.

4.3.4 DBMS-Kernel Integration

The UB-Tree utilizes an underlying B^+-Tree, therefore one dimensional bulk loading techniques can be reused. This makes it easy to integrate the UB-Tree [RMF+00] as well as UB-Tree bulk loading. It is only necessary to extend existing loading tools by Z-address calculation and page splitting for UB-Trees.

This means there are two simple changes in the bulk loading tools having the architecture depicted in Figure 4.3. The affected parts are designated by stripped boxes. The first part of the loading process (key calculation) requires adding Z-address calculation for UB-Trees. This should be nothing more than adding a new key calculation function. The other change is in the third part of the loading process. Here, it is necessary to use the UB-Tree page splitting algorithm in order to get well formed regions, i.e., regions which are as rectangular as possible, since this affects the query performance.

Thus, it is possible to use existing loading techniques, e.g., for Transbase HyperCube, the first DBMS with integrated UB-Tree, these changes have been made without changing the actual loading algorithm of the Transbase loader.

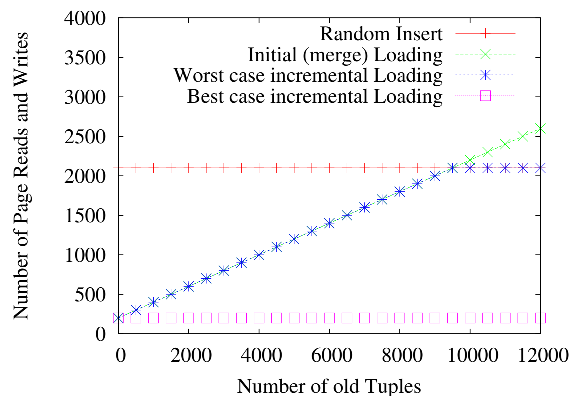
4.3.5 Performance Analysis

The complexity of external sorting in general has been discussed in [Knu73, Agg00].

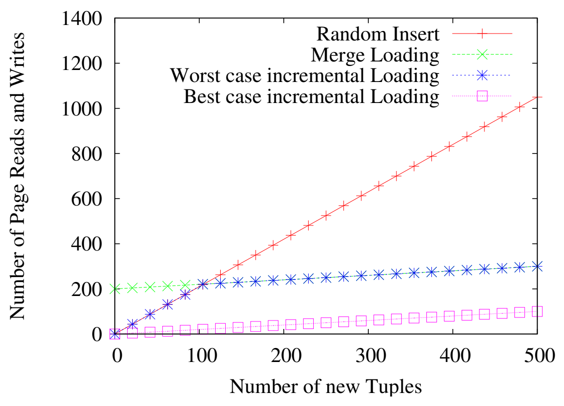
With merge sort in practice, sorting a file of P pages with n tuples is linear and requires P reads for the input data, P writes for the initial runs, P reads for the initial runs and P writes for the resulting output data. This makes $2P$ sequential reads and $2P$ sequential writes. CPU cost divides into key calculation, which has also $O(n)$ plus the cost for the used internal sorting algorithm e.g., worst case for heap sort is $O(n \log n)$.

Concerning only the I/Os of data pages without caching we can make the following worst case estimations for the number C of page reads and writes, where M is the number of tuples which fit onto one page, n the number of new tuples and o the number of old tuples, which reside already in the index. Index pages are neglected, because they usually can be cached.

random insertion: $C = \frac{n}{M} + 2n$, because $\frac{n}{M}$ pages are read clustered from the input file and one page has to be retrieved and stored from the existing UB-Tree for each new tuple.



(a) With 1000 new tuples



(b) With 1000 old tuples

Figure 4.4: Estimated Number of Page Reads and Writes
for pages with a capacity of 10 tuples

initial loading: $C = 2\frac{n+o}{M}$, because $\frac{n}{M}$ pages of data are read clustered from the input file and $\frac{o}{M}$ pages are read from the existing UB-Tree, while $\frac{n+o}{M}$ pages are created.

incremental loading: Worst case is $C = \frac{n}{M} + 2n$, when each input tuple belongs to another page. This can happen only when the number of input tuples is equal or less than the number of pages in the existing UB-Tree. The best case happens when there are few or no updates of existing pages. It is $C = 2\frac{n}{M}$, since $\frac{n}{M}$ pages are read clustered from the input file and $\frac{n}{M}$ pages are newly created.

When using $\frac{o}{M}$ we assume that the existing pages are filled to 100%. Of course this might not be the case and then leads to even worse results. Page utilization ρ can be taken into account by using $\frac{o}{M\rho}$, however for reasons of simplicity we neglect it here. In Figure 4.4 we have made some plots which show the expected loading cost.

Figure 4.4(a) shows a plot for inserting 1000 new tuples to a certain number of old tuples, while Figure 4.4(b) shows a plot for inserting a certain number of new tuples to 1000 old tuples. It shows the I/O-complexity of the different loading techniques depending on the number of old resp. new tuples.

Depending on the relation of the number of old and new tuples C can be simplified. The cost for random insert is independent on the number of existing tuples, since we have to read and write exactly one data page for each tuple. With $o = 0$ or $o \ll n$ we get $C = 2\frac{n}{M}$ for initial loading resp. initial merge loading, this means the loading time depends only on the number of new tuples. For $o \gg n$, we get $C = 2\frac{o}{M}$, i.e., the loading time only depends on the number of old tuples. For incremental loading it is a bit more complicated, because we have to take into account if new tuples contribute to existing pages of the UB-Tree or to new pages. In worst case, when each tuple contributes to a different existing page, we get the same performance as for random insertion. However, when new tuples mainly contribute to new pages, we get a performance which is similar to initial loading of a new

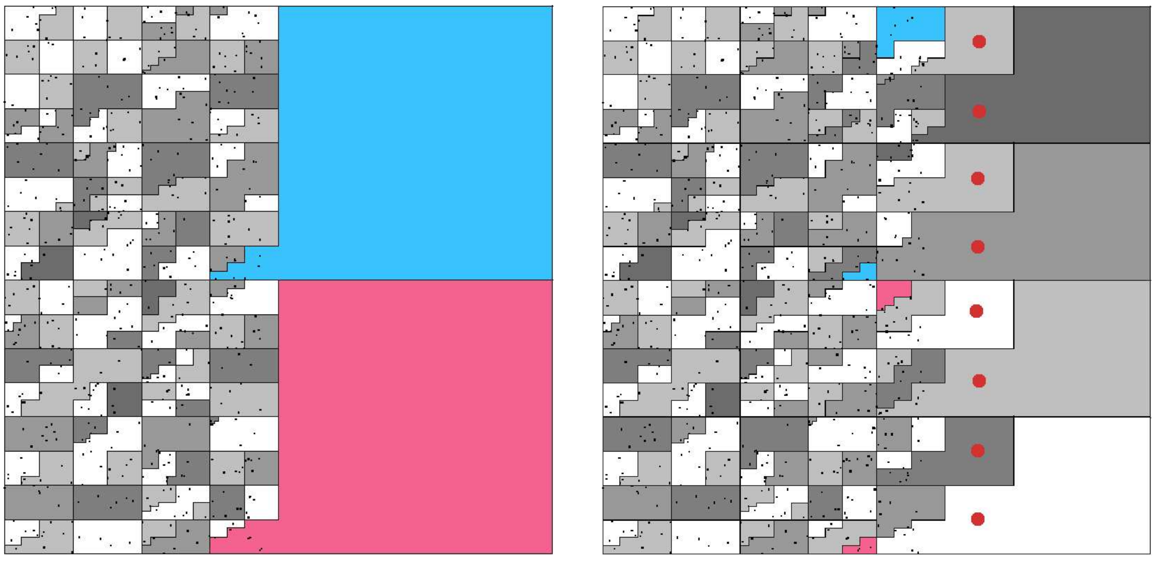


Figure 4.5: Changes of UB-Tree Space Partitioning during Incremental Loading of a new data set into an unpopulated part of a two dimensional universe

UB-Tree.

Figure 4.5 depicts this case where loading new data contributes to empty parts of an existing UB-Tree. This affects usually only a limited number of pages and therefore only a few pages of the existing UB-Tree have to be updated. In the example we have loaded a new data set into the area right to the middle of the two dimensional data space. This requires only two updates of existing pages (the two colored regions on the right in Figure 4.5(a)), while the majority of pages are newly created ones. Another incremental load further right would require 8 page updates (red points in Figure 4.5(b)). The necessary updates can be estimated by the cost functions for UB-Tree presented in [Mar99].

Time is always one dimensions of a data warehouse and older data usually will not be modified after loading, thus new data contributes to a part of the cube which contains no data so far. Therefore, incremental loading is superior compared to initial merge loading, which has to merge a huge existing database with a relatively small new data set.

4.3.6 Performance Evaluation

In cooperation with Teijin, researching on OLAP server applications, we have implemented the presented loading algorithms for the UBAPI. The architecture of the implemented tools is shown in Figure 4.6.

In order to avoid measurement deviations, all measurements have been performed on designated test computers only running the necessary processes.

Clustered UB-Trees with a guaranteed page utilization are best for range query performance. Therefore, we do not present measurements of range queries, but only the results

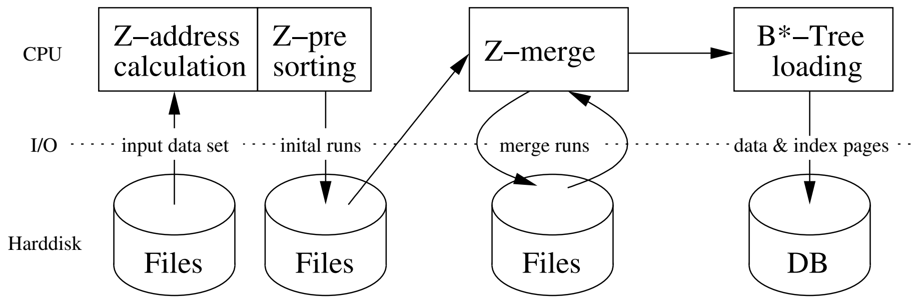


Figure 4.6: The Bulk Loading Architecture as implemented for UB-Trees

on the clustering resp. the number of updated pages for incremental loading. See also [RS81] for a discussion of time and space optimality in B$^+$-Trees.

The presented results are also an indicator for the quality of the spatial clustering of the Z-curve, i.e., the better the clustering the fewer existing regions are affected when loading new data into unpopulated parts of the universe.

4.3.6.1 Artificial Data

The used machine for this measurement was a Sun Ultra 1 with a 167 MHz CPU and 64MB RAM. The secondary storage medium was an external IBM Ultrastar 18XP hard disk with 18GB storage.

We have used a data warehouse cube according to one used at Teijin, which has the four dimensions *Customer*, *Organization*, *Product*, and *Time*. The data distribution was uniform and we loaded two different data sets. One with 50000 (50k) tuples and another one with one million (1m) tuples. The tuple size is 32 bytes and 56 tuples fit on one page with a size of 2kB.

The measurements compare the following cases:

1. Loading of binary vs. ASCII format of temporary files,
2. loading of presorted by *Time* vs. random input data set and
3. the number of created pages of initial loading vs. incremental loading

Table 4.6 shows the comparison of binary vs. ASCII format for temporary files for random order. Since only a UB-Tree for the same universe delivers the needed Z-order, we neglect the order of input data[1].

The loading time was measured in seconds. This yields that using a binary format will result in $\approx$ 30% shorter loading time. For bigger data sets it should be even faster, since fewer merge runs are necessary. This is clearly visible in case of the 1m-random measurement decreasing the loading time by 60%.

[1][Zir03] discusses an algorithm for creating a UB-Tree from an ordered input. While the algorithm usually does not require external sorting, it cannot guarantee a given page utilization.

# of tuples	Loading Time		Page Numbers	
	ASCII	Binary	Initial	Incremental
50k	55s	39s	1191	1235
1m	2259s	939s	23810	24537

Table 4.6: Input Data Format and Page Number Comparison for Artificial Data

Table 4.6 shows the number of pages, which have been created by initial loading resp. incremental loading. The input data has been split up into 7 time periods. In case of this artificial data there are about 3% more pages with incremental loading. However, building the complete 1m DB with one initial and 6 incremental runs took 1722 seconds with incremental loading, while merging the existing UB-Tree and the new data sets with initial merging loading took 6280 seconds. For larger data sets the performance of initial merging loading will be even worse.

4.3.6.2 GfK Data Warehouse

The used machine for this measurement was a Sun Enterprise Server 450 with two Sparc-Ultra4 248 MHz CPUs and 512MB RAM. The secondary storage medium was an external 90 GB RAID system. UBAPI was used as UB-Tree implementation.

In order to evaluate our algorithms with real world data we have loaded a data warehouse from a leading German consumer-market analyst institute (GfK). Detailed informations about the GfK-DWH as used for the tests in this thesis can be found in the Appendix Appendix B.1.

The data is already pre-aggregated to two month periods. Some data sources do not deliver data in a finer granularity, but it also reduces the data volume that must be handled. The DWH has three indexed dimensions *Product*, *Segment*, and *Period*. The dimension hierarchies have been modeled with multidimensional hierarchical clustering (MHC) [MRB99, Pie03], which is crucial for range query performance as it guarantees clustering of data w.r. to the dimension hierarchies.

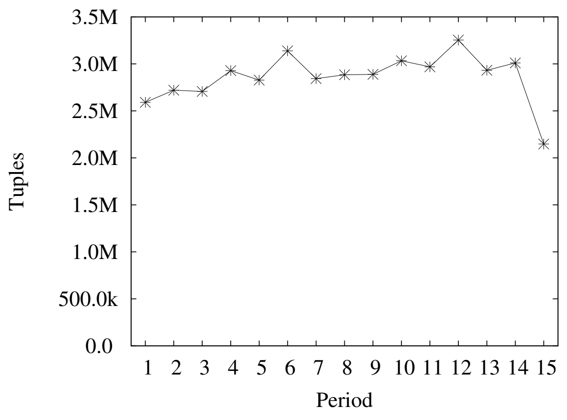
The snapshot of the data warehouse used for our measurements consists of $\approx$ 43 million fact tuples from fifteen two-month periods. In our measurements we have only considered the fact table, since this is the biggest table and new data contributes mainly to this table. The source data was stored in ASCII flat files and the binary representation of one tuple has 56 bytes. The page size was 2kB, but due to page management data only 31 tuples per page could be stored. Page utilization was set to 78%, which results in 24 tuples per page.

The periods have been loaded one by one into a UB-Tree. The first period was loaded with initial loading and the subsequent ones with incremental loading resp. initial loading merging. Random insertion was not considered, because it is not competitive.

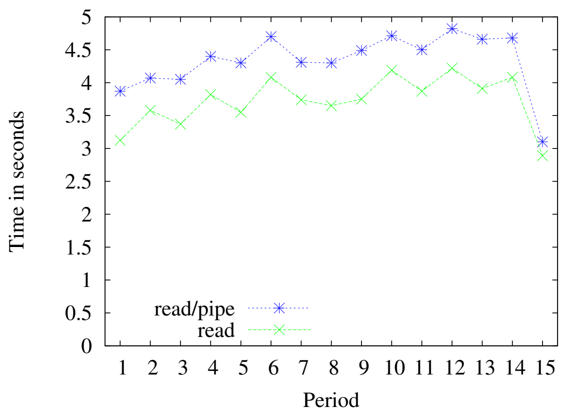
Figure 4.7(a) depicts the data distribution w.r. to the periods, which is directly reflected in the processing times. Figure 4.7(b) shows the time required to read and read+pipe the input file. It is only a small fraction, below 0.5%, of the time required for address

Loading Type	Final Number of Pages	Page Utilization
Random Insert worst	2857860	50%
Random Insert B$^+$–Tree	2041328	69%
Random Insert B*–Tree	1714716	81%
Initial Loading	1786163	77%
Incremental Loading	1790899	75%

Table 4.7: Expected Page Numbers for Initial and Incremental Loading the GfK DWH

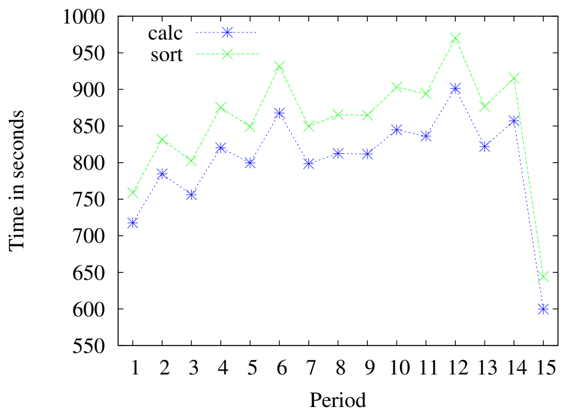


(a) Data Distribution

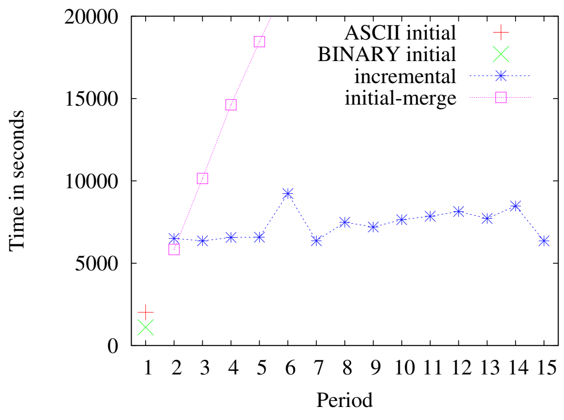


(b) Read and Read/Pipe

Figure 4.7: Data Distribution and Times for Copying resp. Pipe-Lining



(a) Address Calculation and Sorting



(b) Loading

Figure 4.8: Address Calculation, Sorting and Loading Performance for GfK DWH

calculation, sorting and loading, since all reads are sequential. Thus also CPU time must be considered.

Further, Figure 4.8(a) shows the time required for address calculation and sorting of the periods. Again the data distribution is directly reflected.

Figure 4.8(b) shows the measured times for loading. For loading period one we have used initial loading with ASCII and binary temporary files. For all other periods the binary format was used. We can see that binary format gains a speedup of a factor of two.

All the remaining periods have been loaded with initial merge loading resp. incremental loading. For loading the second period the two algorithms perform similar while incremental loading is slightly slower due to queries on the existing UB-Tree (see Figure 4.5).

For the subsequent periods incremental loading is faster, because it only depends on the number of input tuples. For example, period 6 has more input tuples than period 5 or 7 and thus a peak is visible in the loading time.

For an integrated version of the loading algorithms one can expect an even higher speedup, because the UBAPI causes a lot of interprocess communication and performs its own page handling causing additional costs for query processing.

The page statistics in Table 4.8(a) on the facing page show that loading one period in average updates 20 pages while creating 119393 new pages. This confirms our assumptions and analysis that loading new data into an unpopulated part of the cube updates only a small portion of the existing pages.

Table 4.8(b) on the next page shows that $\approx$99.98% of the pages of the final UB-Tree have an utilization of 74% resp. 77%. The remaining $\approx$0.02% pages have an average page utilization of 65%. The total number of pages for random insertion in worst case and on average expectation will vary between 2041328 and 2857860 depending on the kind of underlying B$^+$-Tree. With initial loading of all periods and a page utilization of 78% = 24 tuples per page, one gets 42867902/24 = 1786163 data pages, i.e., there are only 4737 = 1790900 − 1786163 additional pages with incremental loading which finally had 1790900 pages.

Figure 4.9(b) on page 86 shows that while the newly created pages are correlated to the data distribution, the page updates in Figure 4.9(c) are not correlated to it. In fact they reflect the split levels, i.e., the quad-tree partitioning w.r. to time, i.e., when inserting before a major split we get more updates than after it. This is depicted in Figure 4.9(d), where major partitionings between the periods with MHC-surrogates 11:12, 13:14, 15:16, ...

Thus incremental loading requires to update only a few pages of the existing UB-Tree as the different periods contribute data to different parts of the cube. A disadvantage for range query can be neglected, as the average page utilization is nearly as high as for initial merge loading. The huge benefit of incremental loading is the drastically reduced loading time when loading new data to an existing cube. This time becomes available for query processing or other maintenance operations.

During the process of modelling a database it should also be taken into account how to cluster the data, e.g., where the application allows, to separate new and old data w.r. to the indexed universe. For the GfK we have used MHC [Pie03] to cluster the dimension

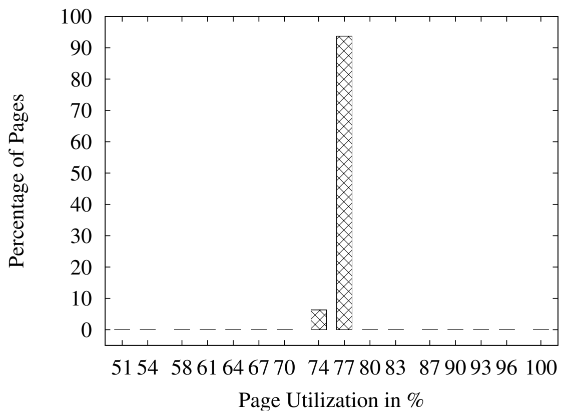
Period	MHC-surrogate	total	updated	created
1	9	112641	1	112640
2	10	225952	17	113311
3	11	338714	32	112762
4	12	460728	8	122014
5	13	578525	34	117797
6	16	709274	1	130749
7	17	827758	32	118484
8	18	947911	18	120153
9	19	1068263	35	120352
10	20	1194613	8	126350
11	21	1318283	34	123670
12	24	1453864	3	135581
13	25	1576039	34	122175
14	26	1701450	17	125411
15	27	1790900	29	89450
total		**1790900**	303	**1790899**
average		–	**20.2**	119393.26

(a) Page updates and creations

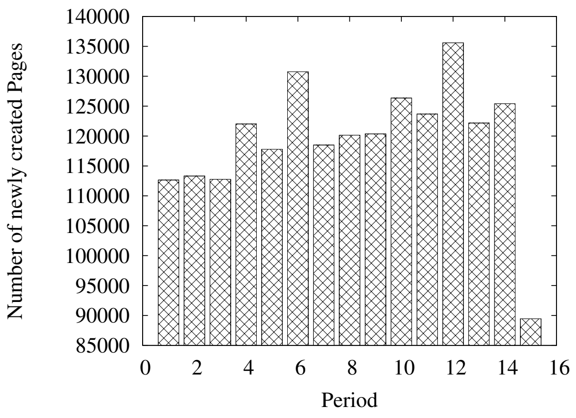
% of all pages	page count	page utilization	tuple number	% of all tuples
0.0018	33	51%	528	0.0012
0.0018	33	54%	561	0.0013
0.0020	36	58%	648	0.0015
0.0021	37	61%	703	0.0016
0.0020	36	64%	720	0.0017
0.0024	43	67%	903	0.0021
0.0024	43	70%	946	0.0022
6.2913	**112670**	**74%**	**2591410**	**6.0451**
93.6913	**1677917**	**77%**	**40270008**	**93.9398**
0.0004	7	80%	175	0.0004
0.0003	5	83%	130	0.0003
0.0005	9	87%	243	0.0006
0.0004	7	90%	196	0.0005
0.0002	4	93%	116	0.0003
0.0003	5	96%	150	0.0003
0.0008	15	100%	465	0.0011

(b) Page distribution w.r. to to utilization

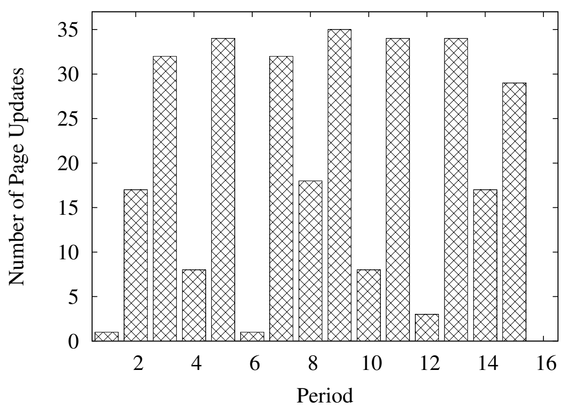
Table 4.8: Page Statistics for Bulk Loading of GfK DWH



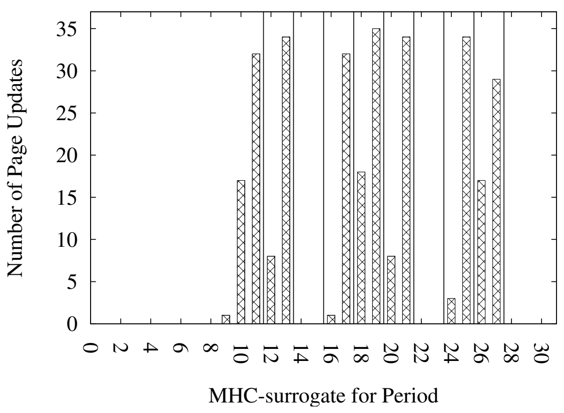
(a) Page Distribution w.r. to Utilization



(b) Newly Created Pages



(c) Page Updates



(d) Page Updates by MHC-surrogate

Figure 4.9: Page Statistics for Bulk Loading of GfK DWH

hierarchies and thus achieve this separation.

4.3.7 Conclusion

In this section, we have considered the problem of bulk loading focusing on UB-Trees. Two bulk loading algorithms are presented which are simple, robust, and provide excellent performance for both loading new data into a UB-Tree and queries on the newly created UB-Tree.

Initial loading provides tuple and page clustering, which lead to optimal range query performance. It can also be used for reorganizing UB-Trees and merging an existing UB-Tree with others or a new data set. When initial loading is too expensive, since it affects only a subset of pages of an existing UB-Tree, we can use incremental loading, which is superior compared to random insertion. It is usually much faster than random inserts and it is able to create partial page clustering, i.e., page clustering within the set of newly created pages.

Incremental loading is always beneficial when new data contributes only to so far unpopulated parts of the indexed space. Data warehouse applications have this characteristic and therefore incremental loading should be used here. Thus, we can give page utilization and clustering guarantees. Of course our techniques are also applicable to bulk loading UB-Trees in general, or for other SFC-orders.

Additionally, we have shown that UB-Tree bulk loading can easily be integrated into a DBMS with B$^+$-Trees. Existing techniques for sorting etc. and the existing infrastructure of a DBMS can be reused. One may decide to make only minor changes in order to provide bulk loading for UB-Trees by extending existing B$^+$-Tree bulk loading code. However, one may also integrate the presented incremental bulk loading algorithm with some more effort, gaining the presented benefits.

4.4 Bulk Deletion

This section discusses the deletion of a set of tuples from a UB-Tree. This operation is performed during bulk updates[2], archiving and migration of data.

Another reason is the maintenance of a window of relevant data. In order to limit the resource requirements old data is removed or archived before adding new data.

The tuples that should be deleted are specified by a predicate, e.g., a set of tuples, a multidimensional range or some arbitrary expression.

Again as for bulk insertion, the goal of bulk deletion is to maintain a given page utilization while accessing only the necessary pages, i.e., at least a 50% page utilization and no duplicate page accesses.

[2]Updates on key attributes are a combination of deletion and insertion. Only updates on non-key attributes can be performed in place, i.e., by a point query and update of the attributes.

The deleted tuples might be dumped or they can be the input of an archiving process, i.e., when migrating data from disk to tape that is no longer relevant for high query performance.

4.4.1 Related Work

A B$^+$-Tree deletion-algorithm is given in [Jan95] and [MO95] proposes some improvements. These algorithms are for deleting a single tuple, but not for deleting a set of tuples based on some predicate. To the best of our knowledge, the efficient implementation of bulk deletes from indexes has not been addressed yet.

[Gra03a] discusses how to partition old and new data that should be loaded into an B-Tree. The basic idea is to prepend a new and unique artificial key-component to the compound key of the new tuples. Thus insertions affect a part of the tree that did not exist before and also queries are not affected until also prepending this new key prefix in the query restrictions. This artificial key enhances (concurrently) loading, but clustering my suffer form not inserting the data at its actual location.

The only paper addressing bulk deletion is [GKK$^+$01]. The authors present bulk deletion in the context of a relation indexed by several indexes. Deletes should be performed on the base table first and then in each index, i.e., deletes do not cascade immediately. Deletion in the base table happens in two steps. In the first step, all data pages are scanned and affected tuples are deleted. In a second step the index is reorganized, empty pages are reclaimed or two nodes of a B$^+$-Tree can be merged. As alternative to the two step approach the index can be adapted on the fly in the first step by a modified version of the "On-line Reorganization of Sparsely-populated B$^+$-Tree [ZS96].

The difference between the bulk deletion in [GKK$^+$01] and the algorithm presented here is, that we are only accessing the necessary pages, but not all data pages and the whole index. Further, we minimize the predicate evaluation by deleting whole pages where possible.

4.4.2 Algorithm

In order to delete tuples we have to locate them, i.e., we have to perform a query returning the data pages that may contain relevant tuples. This can be accomplished with the NJI and NJO algorithm as discussed before. After finding a page, tuples are deleted and underflows are handled in order to guarantee a page utilization of at least 50%.

The generic range query algorithm described in Table 3.2 on page 47 can be extended to take deletion and underflow into account. Deletion of the tuples is straight forward, and if the page is empty after deletion it can be removed from the index. Handling underflows of a node is done by redistributing tuples with neighboring pages.

In case of a bulk loaded UB-Tree, both, the previous and the next page, can be read with minimal or no extra physical I/O as they are neighboring pages to the currently processed one and are in the disk and OS caches already. The fundamental difference w.r. to the tuples on those pages is that those tuples on the previous page are not subject to deletion anymore, but those of the next page may be deleted.

Moving tuples to the next page might cause processing them again, when the NJI falls on the next page. As those moved tuples are at the start of the page we can avoid this by bookkeeping the number of tuples moved to the next page. But there is another drawback, i.e., when deletion on the next page causes an underflow, those tuples have to be moved again. Further, when redistributing tuples from the next page to the current one we need to make sure they are not subject to deletion, i.e., we need to apply deletion to the next page before moving, but this might cause an underflow on the next page.

Therefore, it is better to redistribute tuples with the previous page, as those tuples are not subject to deletion anymore. The location of the previous page can be gathered with little extra cost during search for the current page, but we do not access it unless required.

The different cases to handle after deleting tuples from a page are:

Page is empty: The page and its separator can be deleted from the index.

Page utilization $\geq$ 50%: Store the page.

Page utilization $<$ 50%: An underflow occurred

- **Page utilization + previous page utilization $\leq$ 100%:** The remaining content of the page is moved to the previous page without causing an overflow there. After moving, the page and its separator can be deleted from the index. The previous page is stored and its separator updated, i.e., set to the separator of the current page.

- **100% $<$ Page utilization + previous page utilization $<=$ 150%:** Move tuples from the previous page to the current one to fix the underflow. Obtain new separator for previous page and store the separator and both pages.

During deletion, the path to the currently processed page is kept in memory and thus subsequent deletions in the index part do not cause further I/O. Underflows are just handled when switching to a new path, thus merging pages is only performed once per index page.

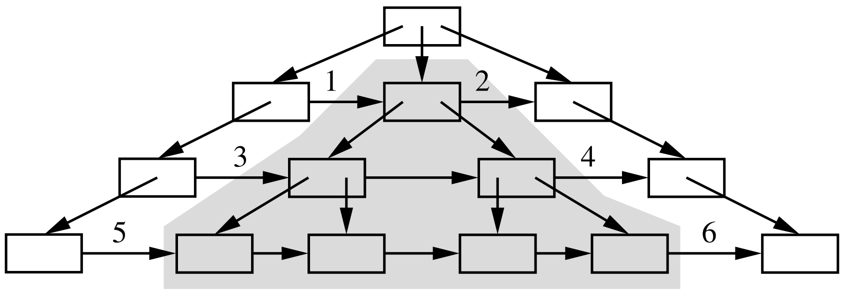
Table 4.9 on the following page shows the algorithm for bulk deletion on data pages. Similar code can be used for the deletion of separators from the index part.

If the page content is completely within a SFC-segment intersecting the query shape, the page can be deleted without inspecting all tuples on it. This is the same technique as discussed in Section 3.2.3 avoiding unnecessary post filtering for range queries. When all remaining tuples of the current page fit on the previous one, the new separator of the previous page is simply the separator of the current page, which avoid unnecessary calculations (line 23). In case of redistribution, a new separator has to be calculated for the previous page (line 27).

The concept of deleting a complete page when it lies within a $[nji, njo]$ interval can be also utilized on the level of index pages. It allows for an efficient deletion of complete subtrees when pages are linked to their next brother on each level (see Figure 4.10(a)).

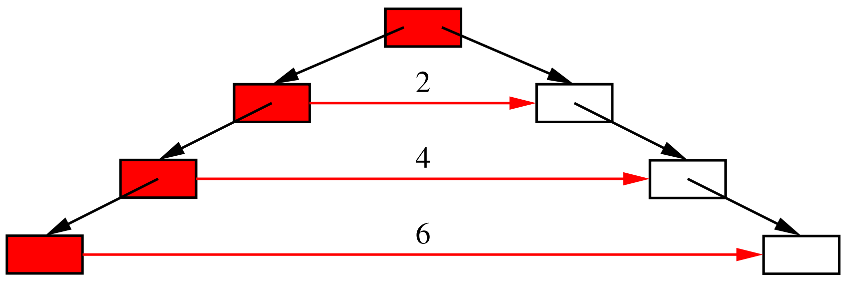
```
void SFCIndex::bulkDelete(Query &Q)
{
  Address nji = S_min(Q), njo, separator;
  DataPageRef page, prevpage;

  while (1) {
    page = findPage(nji);
    separator = page.separator;
    njo = NJO(Q, nji);

    if (nji <= S(page.first()) && S(page.last()) <= njo) {
      page.prevpage.next = page.next;
      deletePageAndSeparator(page);
    } else {
      page.removeTuples(Q);
      if (page.utilization() == 0) {
        page.prevpage.next = page.next;
        deletePageAndSeparator(page);
      } else if (page.utilization() < 0.5 && page.previous) {
        // findPage() also should have located the previous page
        prevpage = fetchPage(page.previous);
        if (prevpage.utilization() + page.utilization() <= 1) {
          prevpage.addTuples(page);
          prevpage.next = page.next;
          deletePageAndSeparator(page);
          prevpage.separator = separator;
        } else {
          redistributeTuples(prevpage, page);
          storePage(page);
          prevpage.separator = calculateSeparator(prevpage, page);
        }
        storePageAndSeparator(prevpage);
      } else {
        // page.utilization() >= 0.5 or no previous page
        storePage(page);
      }
    }

    if (separator >= S_max(Q)) break;
    nji = NJI(Q, separator);
  }
}
```

Table 4.9: Bulk Deletion Algorithm

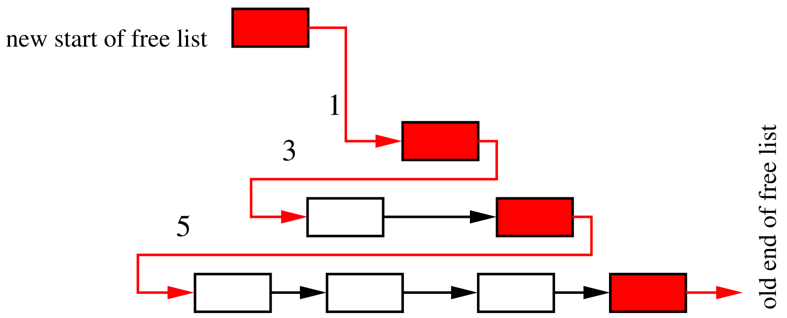


(a) Tree Structure

In the following the links taken from the old tree structure are numbered as they are shown there.



(b) Transformed Tree



(c) Free List

Figure 4.10: Deleting and Freeing Subtrees

Free pages are managed as linked list, called the *free list*. Deleting the subtree is adding it to the free list. The root of the subtree becomes the start of the free list and the last page of the subtree points to the old start of the free list (Figure 4.10(c)). The next-links at the end of each level of the subtree are adjusted to point to the first page on the next level (Figure 4.10(c)). The path of pages just before the subtree also requires adjustments (Figure 4.10(b)), but it also provides the links to the first page on each level of the subtree. Thus, adjusting the next-links of the subtree can be handled in parallel by descending both paths at the same time and swapping the next-links of the pages as depicted by the numbers of the links in Figure 4.10.

Finally the index entry referring to the root of the subtree is deleted and possible page underflows are handled. The resulting tree is still balanced and a valid B^+-Tree resp. UB-Tree.

The complexity of a subtree deletion is $O(2h' + 1)$ where h' is the height of the subtree and thus $2h'$ represents the two paths and the additional $+1$ stems from the index page containing the link to the root of the subtree.

In Figure 4.10(b) and 4.10(c) pages that need to be read and written are filled.

4.4.3 Analysis

The algorithm terminates as we are moving forward from the minimum value within the query shape to its maximum by utilizing the NJI algorithm and the page separators related to a nji. This makes the algorithm also cache aware, i.e., pre-fetching is optimally exploited for bulk loaded databases.

The memory and cache requirements are minimal: we only need to cache the path from the root page to the currently processed data page plus the previous page. B^+-Trees in current applications have a height of 3–5, thus the requirements are very small.

The maximum number of accesses to a previous page is bounded by its capacity, i.e., the maximum number of accesses to the previous page occurs when moving one tuple per processed page to the same previous page which initially had an utilization of 50%. Thus in worst case there are $1 + \frac{capacity}{2}$ accesses to a single page, one for processing deletion on the page and the other for adding tuples to it where each tuple comes from a different page. In order to minimize the physical I/O, page writes should be delayed until the previous page is no longer required. A "last recently used" cache policy guarantees this.

When redistributing tuples with the previous page, we should not balance the tuple numbers, but just move enough tuples to the current page in order to gain a 50% utilization again. This maintains the highest possible utilization on the previous page. Further, it leaves space on the current page that can be used for the following pages. This increases the possibility to be able to delete a whole page by moving its content to the previous page instead of updating it.

It can be necessary to run bulk operations exclusively, in order to avoid performance degeneration and locks of query processing. In case of the bulk algorithms presented in this thesis, it is not necessary to lock the whole tree instantly, but while the algorithms proceed they lock modified pages. In the worst case, this locks the complete table/tree, i.e., if tuples contribute to resp. are deleted from all data pages. However, the compete lock will just occur towards the end of the processing, thus leaving opportunities for concurrent operations working on so far unlocked pages.

Removing a whole subtree will only modify few pages ($2h$ pages), but most pages will be deleted completely. This is similar to loading a period of the GfK DWH, as it presented in the last section.

Bulk loading and bulk deletion complement each other and allow to efficiently maintain a window of interest on the data, i.e., only keep the currently important data in the active system while moving old data to tapes, etc.

4.5 Summary

This chapter has introduced the basic concepts of the UB-Tree as required for this thesis. Further, we have presented novel algorithms for incremental bulk loading and bulk deletion. For bulk loading we also present measurements with a real DWH and as bulk deletion is the inverse of it the results should also hold qualitatively for deletion.

Chapter 5

Bitmap-Indexes on Clustered Data

Bitmap indexes [CI98, WB98] are secondary indexes, which are widely accepted as indexing solution for decision support systems (DSS), data warehousing (DWH) and Online Analytical Processing (OLAP). Boolean operations are efficiently supported by them, but they have several severe drawbacks:

- They are secondary indexes and thus non-clustering. Range queries usually cause random page accesses on the indexed base table.
- Their maintenance, during insertion, deletion and updates is extremely costly, i.e., these operations are not local, but they affect large or all parts of a bitmap. Thus bitmap indexes are primarily used for read-only applications.
- They[1] are only suitable for dimensions where the cardinality of actual attribute values in a relation is low, since one bitmap is required for each attribute value appearing in the relation.

Several commercial DBMSs feature bitmap indexes. Starting with the pre-relational *Model 204* [O'N89] also RDBMSs incorporated this technique, i.e., RedBrick [Ede95], Sybase [Ede95, Inc97], Informix [Pal00, O'N97], Oracle [Ora99] and IBM [Win99]. Commercial DBMSs lacking built-in multidimensional indexes, but providing bitmap-indexes can benefit when clustering the base table according to a SFC. This can be achieved by sorting the data according to a SFC address before loading it into the DBMS. Thus, no modification of the DBMS are necessary as sorting and address calculation can be performed by application code.

As updates to already inserted data do not occur or are extremely rare, clustering will not be deteriorated much. Clustering within a set of jointly loaded data is optimal, just between different loading sets it will not be maintained. In order to maintain clustering also for update one can add the SFC-address to the tuples and index it by a clustering index.

[1]This hold for bitmap indexes encoding attribute equality. See the following section on encodings.

Clustering the base table can significantly reduce the creation cost and size of bitmaps and increase the query performance. This chapter discusses the effects on bitmap indexes caused by clustering the indexed base table. In order to allow a better understanding and explanation of effects, we provide an introduction to the basic concepts of bitmap indexes.

5.1 Introduction

Bitmaps are bit-strings storing which rows of a table satisfy a given predicate. Each bit of the bit-string *encodes* whether the given predicate is true or false for the row corresponding to the bit position. *Compression* is used to reduce the memory requirements of bitmaps and thus save I/O and main memory at the expense of a higher CPU load.

The set of all existing bitmaps for an attribute A is called *Bitmap Index* on attribute A. In the following we will discuss *encoding* and *compression* in detail.

5.1.1 Encoding

A bitmap is a bit-string where each bit-position corresponds to a row of the indexed table and the value of the bit encodes if the row satisfies a given predicate or not. Standard bitmap indexes encode if a given row has a given attribute value or not, thus there is one bitmap for each occurring attribute value. These bitmap indexes are called *value encoded bitmaps.*

A generic definition of bitmaps is the following:

Definition 5.1 (Bitmap)
We call a bit-vector $B_\rho(R) = [b_0 b_1 \dots b_n]$ *bitmap* for the predicate ρ on relation $R = \{\vec{t_0}, \cdots, \vec{t_n}\}$, where $b_i = \rho(\vec{t_i})$ and $n = |R|$. $\diamond$

Any combination of predicates suitable for query processing might be used. Actually, most implementations are only offering one predicate type, i.e., a bitmap for each attribute value where the bitmap encodes if a row has an attribute value equal to the attribute value encoded by the bitmap.

Definition 5.2 (Bitmap-Index)
We call $\mathbb{B}(R) = \{B_{\rho_0}, \cdots, B_{\rho_m}\}$ *bitmap index* on relation R with the predicates $\rho_0 \cdots \rho_m$. $\diamond$

Bit-positions can be directly mapped to row-ids. In case of non persistent row ids, bit-positions have to be mapped to rows by a vector to generic tuples identifiers (TIDs), e.g., keys of a primary index. This has to be stored only once for all bitmaps of a given table as the bit-positions of all bitmaps are referring to the same tuples.

Example 5.1 (Encoding Bitmaps)
Table 5.1 shows a simple example for a *value encoding bitmap index* with the three attributes *Product*, *Customer*, and *Year* in the left table and six example rows.

Product	**Customer**	**Year**
TV	R	03
CD	R	02
TV	F	01
PC	R	01
PC	V	02
PC	M	02

(a) Relation

Product			Customer				Year		
TV	**CD**	**PC**	**R**	**F**	**V**	**M**	**03**	**02**	**01**
1	0	0	1	0	0	0	1	0	0
0	1	0	1	0	0	0	0	1	0
1	0	0	0	1	0	0	0	0	1
0	0	1	1	0	0	0	0	0	1
0	0	1	0	0	1	0	0	1	0
0	0	1	0	0	0	1	0	1	0

(b) Bitmaps

Table 5.1: A Simple Relation and its Attribute-Value Bitmap Indexes

The table on the right hand side shows the corresponding bitmaps, i.e., one for each occurring attribute value. *Product* has the three distinct values `TV`, `CD`, and `PC` thus there are three bitmaps, etc. The bitmaps contain a `1` for those rows having this attribute value and a `0` for those that have not. ◇

Depending on the query profile and data distribution other encodings might be useful. The most popular encodings beside the presented one are *range encoding* and *interval encoding* bitmaps. Each bitmap encodes whether a row's attribute value falls within a given range. Thus instead of a bitmap for each attribute value we have only bitmaps for each range which reduces the space requirements of high cardinality dimensions [Kou00, CI99].

However, the drawback is, that results of the bitmap evaluation are a superset of the actual result. Further, choosing the ranges to guarantee optimal performance is non trivial and depends on the query profile and the data distribution, thus it is not possible to provide reasonable worst case performance guarantees anymore.

Another application of *range encoding* is *hierarchical encoding* [WB98], i.e., the bitmaps encode the nodes of the hierarchy by ranges on the hierarchy attributes. DWH-queries often use hierarchies for the navigation (drilling) and restriction and thus this encoding works well for DWHs.

In order to solve the problems of *value encoded* bitmap indexes for high cardinality domains, [Inc97] has introduced *Bit-Wise* bitmap indexes, i.e., a bitmap encodes if a specific bit in the attribute value is true or not. This reduced the number of bitmaps to the number of actually use bits in the binary representation of the attribute, e.g., for an attribute of type integer with four bytes, bitmaps are only created for those of the 32 bits actually set in any attribute value stored in the relation.

Thus, also ranges restrictions can be handled with only a small number of bitmaps. However, run-lengths are reduced as there are more bits set in the bitmap and thus compression (See Section 5.1.3) suffers.

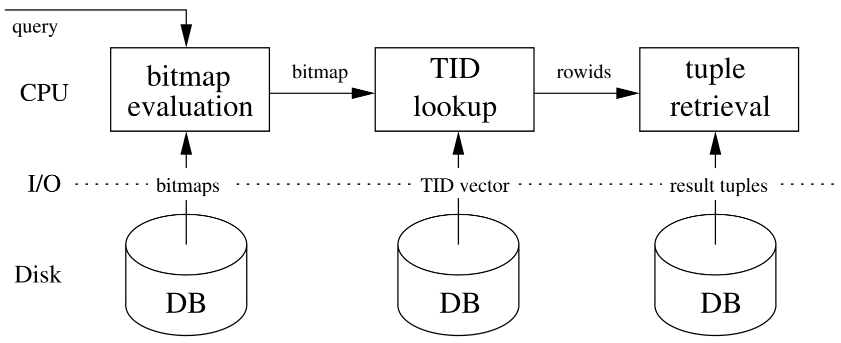


Figure 5.1: Query Processing with Bitmap Indexes

5.1.2 Query Processing

In order to perform a query for all tuples having a given attribute value, all rows are fetched which are indicated by a 1 in the bitmap for this value.

More complex predicates require to combine bitmaps according to the predicate, which is usually a boolean operation, e.g., a bitwise `AND` or `OR`. For example, when looking for all tuples containing an attribute value from a given set of values, the bitmaps corresponding to the values in the set are merged by a bitwise-`OR` operation. The resulting bitmap determines the result rows.

Example 5.2 (Range Queries with value-encoded Bitmap Indexes)
In order to process a range query with bitmaps, bitwise `AND` and `OR` operations are necessary. First, for each dimension the bitmaps within the range restriction on the dimension are `OR`ed resulting in a bitmap per dimension. These bitmaps designate all the tuples w.r. to the restriction on a single dimension. Next, these bitmaps are `AND`ed resulting in the final bitmap that is used to fetch the result tuples. ◇

Before retrieving the result tuples, the positions of the "1" bits in the result bitmap have to be mapped to TIDs via a vector or directly to row IDs, giving a direct reference to a page and the tuple position on the page. The whole process is depicted in Figure 5.1.

The first two steps cause only sequential reads, i.e., reading the bitmaps and the TID-vector utilizes pre-fetching, but this does not hold for the last step.

As for all secondary indexes the access to the actual result tuples is performed by random access (see also Section 2.5.3). It either uses the keys of the primary index organizing the base table or the row identifiers directly encoding the page (and position) containing a tuple. To some extent random accesses can be reduced by sorting the result TIDs before accessing the base table, but this has the drawback of being a pipeline-breaker as sorting can start just after all TIDs have been obtained. Furthermore, sorting might become an external operation causing I/O when the resulting TIDs do not fit into the main memory anymore.

Thus mapping bit positions directly to row IDs is better w.r. to the direct access while being not so flexible in case of updates, i.e., if the location of tuples changes and thus the row ID become invalid while TID remain valid.

5.1.3 Compression

Bitmaps are not stored as a set of plain bit-strings, but they are compressed in order to reduce their space requirements. This reduces I/O, allows for caching more bitmaps in main memory and even may speed up boolean operations. However, there are also drawbacks:

- Higher cost during query processing if decompression is necessary in order to perform boolean operations. CPU operations may quickly become the bottleneck, as bitmaps are read by efficient sequential reads.
- Due to compression updates become very costly, as they require reading and decompressing the whole bitmap, changing the affected bits and finally compressing and writing the compressed bitmap back to disk. Starting from the position of the first affected bit the remaining bitmap needs to be re-compressed and re-written again. Changes in the compressed bitmap affecting literal sequences are an exception to this.

Run Length Encoding (RLE) is commonly used to compress bitmaps. RLE is local in comparison to dictionary compression techniques like *Lempel-Ziv-Welch* (LZW) [Wel84] compression, i.e., it does not require dictionary lookups for (de-)compression and thus is superior w.r. to compression and decompression speed. However, it does not honor repeating sequences of varying length, resulting in a poorer compression. [Joh99] performed extensive tests comparing the byte-aligned RLE and LZW compression of bitmaps and came to the conclusion that there is no significant advantage of LZW w.r. to the compression ratio. Further, RLE allows boolean operations to be performed with the compressed representation, so there is no need to materialize the bitmaps to perform the operation.

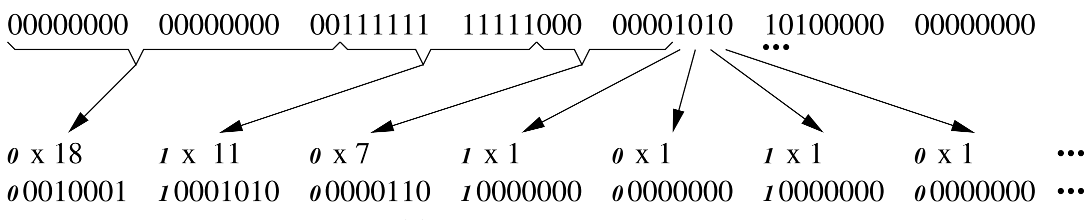
RLE partitions a bitmap into chunks of the same length, i.e., a single bit, a byte or a set of bytes. A sequences of chunks with equal values is called *run*. It is encoded as the chunk value and the length of the run.

Sequences longer than their encoding are compressed, but sequences shorter than their encoding will waste memory. The memory waste can be reduced by reserving a bit that encodes if the remaining bits encode a run or a literal sequence. Consequently, the run length should be maximized in order to gain the better compression.

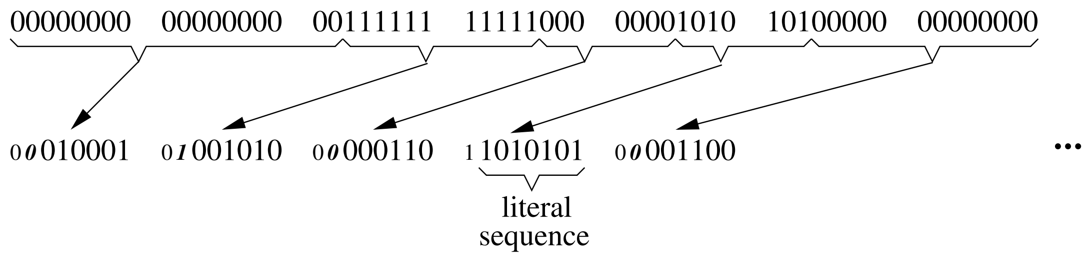
Example 5.3 (Compressing Bitmaps)

Using a bit as chunk size, one byte can be used to represent runs of a maximum length of 128 bits, i.e., one bit is used to store the bit value and the remaining 7 bits store the run length as shown in Figure 5.2(a).

If a run gets shorter than 8 bits then space is wasted, e.g., a run with length one is encoded with a whole byte thus requiring eight times more space. One can reduce



(a) Run Length Encoding



(b) Run Length Encoding with Escapes for literal sequences

Figure 5.2: RLE Bitmap Compression

this by reserving another bit that determines whether the remaining bits are encoding a run or a literal sequence (the first bit in Figure 5.2(b)). In case of a run the next bit encodes the bit value and the remaining six bits the length of the run. As there are no runs of length zero it is sufficient to store ($length - 1$) in the remaining bits increasing the possible run length by one, i.e., 0 encode a run-length of 1, ... ◇

Real world data usually is highly skewed, i.e., non-uniformly distributed, and thus the worst case of run length distribution usually does not occur. Furthermore, often there is a relation between different dimensions. For example products are only sold for a given period and then they are replaced by newer ones, thus bitmaps on *Product*/*Time* benefits from clustering the data according to both dimensions.

Example 5.4 (Maximizing the run-length)

Consider a relation R with n tuples $R = \{\vec{t_0}, \cdots, \vec{t_n}\}$ and the two subsets $T = \{\vec{t_i} | \rho(\vec{t_i}) = 1\}$ and $F = \{\vec{t_i} | \rho(\vec{t_i}) = 0\} = R \setminus T$ with $i = [0, n]$.

Assuming a uniform distribution of tuples w.r. to $\rho(\vec{t_i})$ and $|T| \leq |F|$ starting with a tuple satisfying ρ one gets $|T|$ "1"-runs with length 1 and $|F|$ "0"-runs with a length of $\frac{|T|-|F|}{|F|}$.

The best possible compression is achieved when all tuples are ordered by $\rho(\vec{t_i})$ then there are only two runs or just a single run if all tuples satisfy ρ in the same way. ◇

A RLE technique very efficient for query processing is the *Byte-Aligned Bitmap Compression* (BBC) [Ant94] encoding runs of equal byte values. Its big advantage during query processing is that the bitmap chunks can be combined by bit-operations without any shift operation as they are already aligned. Decompression degenerates to perform the boolean operations on a chunk the number of times it repeats and no materialization of the actual bitmap is necessary.

Estimating the size of bitmaps is complicated as it depends highly on the data distribution and the clustering of the data. We are only able to provide best and worst case bounds, which both usually do not occur in real world applications.

5.1.4 Maintenance

Maintenance operations like insertion and deletion are problematic with bitmap indexes as their effects are not local. Due to compression, changes in a bitmap itself require three steps:

1. Reading + Decompression
2. Bit modifications
3. Compression + Write

Further the operations may affect not only a single bitmap, but all bitmaps of a table. In the following, we discuss the required actions for different maintenance operations for attribute encoded bitmaps, as only these are actually available in Oracle.

Insertion: Inserting a new row affects all bitmaps by inserting a new bit at the corresponding position in each bitmap. If the new tuple contains an attribute value that has not been used so far then a new bitmap has to be created for the new value. This does not require reading the base table again, as none of the old rows contain the new value. Furthermore, the TID of the new row has to be added to the TID-vector.

Appending: Appending a new row is a special case of insertion. It is much cheaper as it only appends to the table, affected bitmaps and TID vector. Instead of modifying each bitmap, only the one corresponding to the attribute value of the new tuple need to be modified. This is possible as all bits from the current end of the bitmap up to the position of the new tuples must be zero bits.

Updates: Just those bitmaps containing the old and the new values are affected. They have to be re-packed and written starting from the position corresponding to the updated tuple. In case of new values also the corresponding bitmaps have to be created. TID changes can be tracked in the TID vector. Without affecting the bitmaps.

Deletion: There are two ways to handle this:

- Remove the bit corresponding to the tuple from all bitmaps and the TID vector. This modifies all components of the bitmap index including the TID-vector.

- Marking the TID as invalid in the TID-vector. This needlessly increases the size of the bitmaps, but on the other hand the TID-vector is only locally modified. This is an option if deletions are rarely or when performing a reorganization from time to time.

Due to this, DBMS vendors advise the user to perform maintenance operations on tables with bitmap as follows:

1. drop the bitmap indexes
2. bulk load the new data
3. recreate the bitmap indexes

With growing database size this is not practical anymore as the cost of dropping and recreating bitmaps is direct proportional to the size of the database and not the size of the new data. Thus additionally, the user is advised to partition the data into smaller sets that can be handled efficiently, e.g., one partition per month.

5.2 Bitmaps on clustered data

DWHs are multidimensional cubes and therefore spatial clustering is the key to good range query performance. As we have seen before, this can be achieved by a space filling curve. Chapter 3 has discussed how SFCs enhance the performance of multidimensional range queries by preserving spatially locality as well as possible.

Bitmap indexes are secondary indexes and thus cannot provide clustering, i.e., they do not organize the base table. The base table can be clustered in any desired order. The order also affects the performance of bitmaps on it, i.e., their size and the number of physical random accesses.

Without a clustering index, data can be sorted according to the SFC-addresses before performing a bulk insertion. If clustering should be maintained for random insertion and updates, one can add an additional attribute, the SFC-address, to the relation and index it with a clustering index, e.g., a B-Tree. Both techniques do not require any SFC support within the DBMS and can be implemented in the application.

Fetching of result tuples is still a random access from the logical point of view, but physically the random accesses are reduced, since data is now clustered on disk.

Further, clustering increases the average run length by storing tuples with equal values near to each other and thus reduces the size of the compressed bitmaps in comparison to unclustered data. For uniform data, the effect is not very significant, but with skewed data the runs of zero-chunks are not fragmented anymore.

Due to the curse of dimensionality, the positive effects of clustering will become smaller with increasing dimensionality. However, this depends on the actual data distribution and dimension domains.

5.3 Evaluation with the GfK DWH

The measurements presented here have been conducted with the GfK DWH, as already used in previous measurement sections. It consists of ≈ 42 million fact rows where each row requires 56 bytes. The three dimensions TIME, PRODUCT, and SEGMENT identify the remaining 11 measure attributes. The data was organized in a star-schema and the hierarchies were modelled with Multidimensional Hierarchical Clustering (MHC) [Pie03]. Detailed informations about the GfK-DWH as used for the tests in this thesis can be found in the Appendix Appendix B.1.

The used DBMS was Oracle 8.1.7i running on a Sun Ultra 10 (300MHz, 512MB main memory) with two external Seagate ST39111A (73.4GB) U160 hard disks. The DBMS page size was set to 8KB.

The initialization parameters affecting the performance for bitmap indexing were all set to 40MB, i.e., `CREATE_BITMAP_AREA_SIZE`, `BITMAP_MERGE_AREA_SIZE`, and `SORT_AREA_SIZE`. This should be sufficient to keep the uncompressed tails of all bitmaps in main memory, i.e., for the 370319 distinct fact tuple values and an assumed 100 bytes for the management of each we get a requirement for 35MB.

In order to get a comprehensive overview on the properties of the different indexes, maintenance as well as query performance were measured. We focus on fact table operations as they are a substantial part ([Pie03, PER$^+$03a]) of the overall cost in query procession. The other processing steps are the same for all indexes, i.e., dimension table access to gain the restrictions on the fact table, result aggregation are omitted here.

The measured index-configurations were a UB-Tree base on the UBAPI implementation (see 4.2.1) on top of Oracle, a Compound index, and seven bitmap indexes with different clusterings of the tuples in the fact table.

Table 5.2 gives an overview on the index configurations and the used abbreviations. BM is a bitmap on the unclustered fact data, i.e., there is no specific order of the tuples in the fact table. BZ has the same clustering as the UB-Tree and thus is the reference for comparison with the UB-Tree. BH uses the Hilbert curve for address calculation where the universe was simply created by using the maximum of the dimension cardinalities as cardinality for all dimensions thus creating universe with equal cardinalities. As this deteriorates the clustering we were also using a Hilbert curve where the values of dimensions with shorter cardinalities were padded with 0s as described in Section 3.1.3.2, which is called BH2. Data that is to be loaded into a DWH is usually sorted according to time, so we also added variants of bitmap indexes where the first sort-key was time. BT is just sorted by time, but there is no specific order on the other dimensions, where BTZ and BTH2 are sorted by time first and then Z-order resp. Hilbert-order. Thus just the tuples within one period need to be sorted and the already loaded tuples are not affected anymore.

5.3.1 Maintenance

We consider maintenance to include all necessary steps in order to make new data available for query processing. In the context of DWH the focus is on bulk operations, as new data

Label	**Index Type**	**Clustering Order**
UB	UB-Tree	Z-order
COMP	Compound Index (IOT)	Compound-Order: Time $\circ$ Product $\circ$ Segment
BM	Bitmap Index	Mixed (Random) Order
BZ	Bitmap Index	Z-Order
BH	Bitmap Index	embedded Hilbert-Order
BH2	Bitmap Index	expanded Hilbert-Order
BT	Bitmap Index	Time-Order
BTZ	Bitmap Index	Compound-Order: Time $\circ$ Z-Order
BTH2	Bitmap Index	Compound-Order: Time $\circ$ expanded-Hilbert

Table 5.2: Overview on the measured Index Configurations

is loaded in daily, weekly, monthly, or in other regular time intervals in order to reduce the load. Furthermore, during loading usually no query processing is allowed in order to guarantee consistent query results and minimize the maintenance time.

Depending on the type of the index all or just a subset of the following processing steps are necessary.

- Data sorting/preprocessing
- Index dropping (when adding new data to an existing relation)
- Data loading
- Index creation/maintenance

Table 5.3 gives an overview on the maintenance operations required resp. measured for the different indexes. The bitmap index *BM* does not require any sorting, but all other indexes do require it. Sorting was measured from sorting the mixed resp. time ordered data to the desired target order. The process was performed by an external program performing the address calculation and a merge-sort working on the ASCII flat files. The input flat file had a size of $\approx$ 2.57GB. Dropping bitmap indexes before loading is not really a requirement, however it is recommended by the manual of Oracle due to the high cost of maintenance, i.e., it is usually much faster to drop and recreate the bitmap indexes.

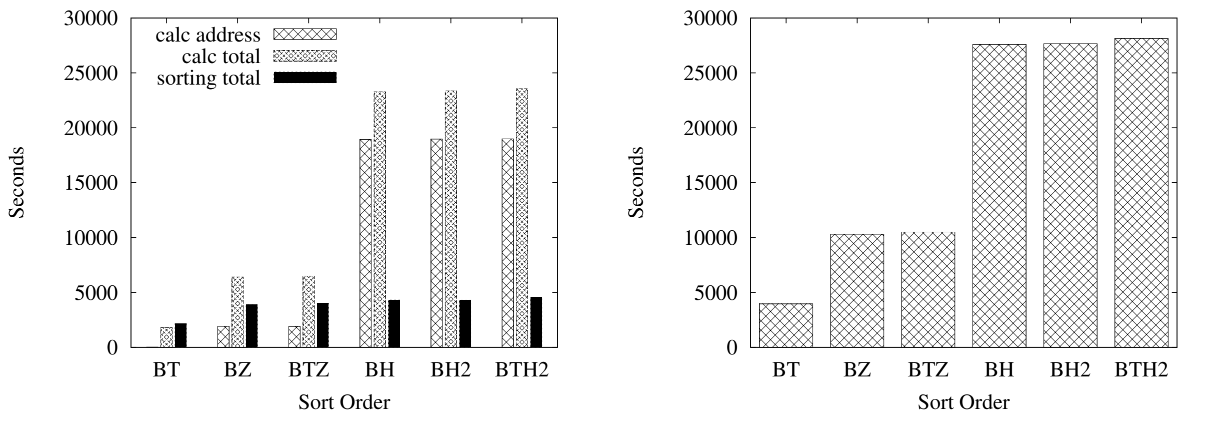
5.3.1.1 Address Calculation and Data Sorting

Before loading new data it has to be sorted according to the desired clustering. As a starting point we used the mixed data, i.e., there was no specific clustering. This was done in two steps:

Operation	**UB**	**COMP**	**BM**	**other Bitmaps**
Data Sorting	+	(+)	−	+
Index Dropping	−	−	+	+
Data loading	(+)	(+)	+	+
Index Creation	(+)	(+)	+	+

+ marks necessary operations, − it not required.
(+) denotes operations that could not be measured independently.

Table 5.3: Overview on the Maintenance Operations



(a) Split-up of costs (b) Sum of costs

Figure 5.3: Cost for Address Calculation and Sorting

1. **Address calculation:** we prepend a new attribute to each tuple which is the sort key. We use a bit-string for this, as this allows for a generic sorting algorithm, i.e., the data-type of the address is always the same independent on the desired order.

2. **Sorting:** Using the prepend sorting-attribute we perform a merge-sort. The results of the address calculation can directly go into the run generation without a need to save them.

Figure 5.3(a) shows the times required to sort the data according to the different orderings. It shows three bars per ordering, where the first is the time spent in the CPU to calculate the address/sort-key, the second the total time for calculation including I/O, the third the total cost for sorting. The overall cost is shown in Figure 5.3(b). The x-axis shows the sorting order according to the names used for the bitmap indexes.

The cost for "T"-addresses is practically negligible, as it is a simple copy operation, thus the 22 seconds required are not visible in the plot. The remaining cost contributing to the total is mainly I/O for reading/writing the tuples and parsing of the input. As the keys of the other orderings are longer they also require more time in total. The calculation of Z-addresses requires 1915 seconds, which is 10 times faster than the Hilbert-address calculation. Using an address calculation algorithm for the Hilbert-curve being not limited to a fixed dimensionality and dimension cardinality makes it so expensive. As we are using a flexible algorithm for the Z-curve, we feel it justified to also use such an algorithm for the Hilbert-curve. The additional cost of the Hilbert-curve calculation stems from adjusting the orientation of the “U”-shapes at each of the 29 levels of the quad-tree.

Whether we pad the values with “0”s or not does not make a significant difference, as this is implemented by simple shift operation before starting the actual Hilbert-curve address calculation.

The longer addresses do affect sorting only during run creation and when reading the runs for the merge phase, but they are not included in the final result. Thus, sorting times are similar for all SFC based orderings, just “T” with its obviously shorter address requires only half the time as it requires fewer initial runs and I/O. In average the “H”-variants require 2.7 times longer than the “Z”-variants and the “Z”-variants 2.6 times longer than “T”.

5.3.1.2 Data Loading and Index Creation

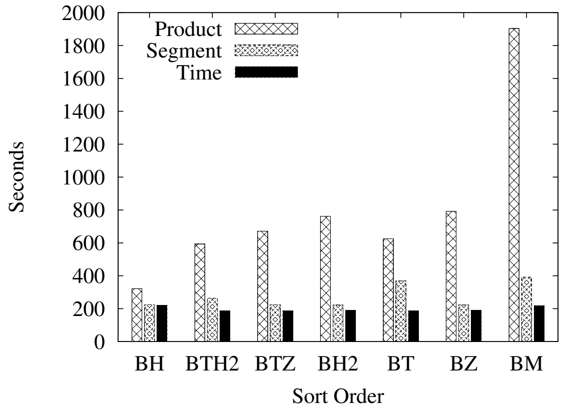
Loading data and creating indexes can be handled together. However, in the case of Bitmap Indexes the Oracle manual advises to drop them before loading and to rebuild them afterwards as maintaining them during loading is too costly.

In case of initially loading the GfK-data into Oracle, it takes in average four times longer to maintain the indexes during loading than to drop them, load the data, and recreate the indexes afterwards.

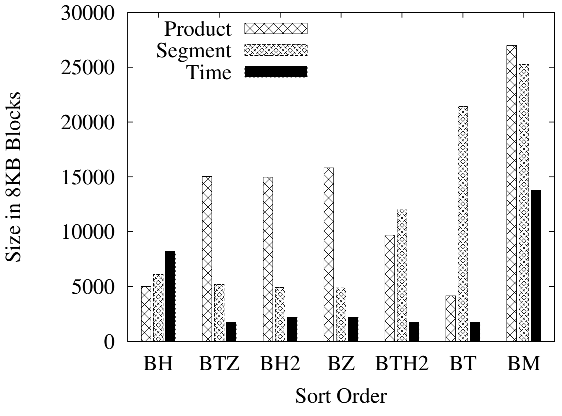
On the other hand clustering indexes like the Compound-Index or the UB-Tree have to reorganize the data according to their clustering. In order to build them efficiently after loading the correct clustering of the base table must be ensured, but in this case the index can also be built without additional cost during loading.

Loading the fact table without an index requires approximately 1400 seconds as this is only copying the data from the flat files into the database. This is the time required by all Bitmap indexes. Loading COMP and building its index requires 4650 seconds and finally the UB-Tree requires 14890 seconds. As UBAPI does not utilize the bulk loader of Oracle but performs random insertion it is that much slower, but with a kernel integration one can expect similar execution times as for COMP with a slight overhead due to Z-address calculation.

Figure 5.4 shows the time for bitmap index creation and the resulting size of the bitmaps per dimension. Only reading the whole table takes 75 seconds, thus, the major cost is the management of the bitmaps. Creating all bitmaps at once does not perform well, e.g., for BZ it takes around 24 minutes in sum to create the 3 bitmap indexes sequentially while it takes more than 2 hours when creating all at once, i.e., with one `CREATE BITMAP INDEX <table>(<rows>)` statement. This probably stems from a bad implementation of the lookup, i.e., maybe Oracle uses too small hash tables. Although increasing the CREATE_BITMAP_AREA_SIZE to 80MB does not improve this but increases the time to



(a) Time for Bitmap-Index Creation



(b) Size of Bitmap-Indexes

Figure 5.4: Maintenance Cost for Bitmap Indexesc

3.5 hours. An own bitmap index creation implementation, which was also used to create the images in Figure 5.5, only required 26MB. We have no explanation why Oracle is not performing better here.

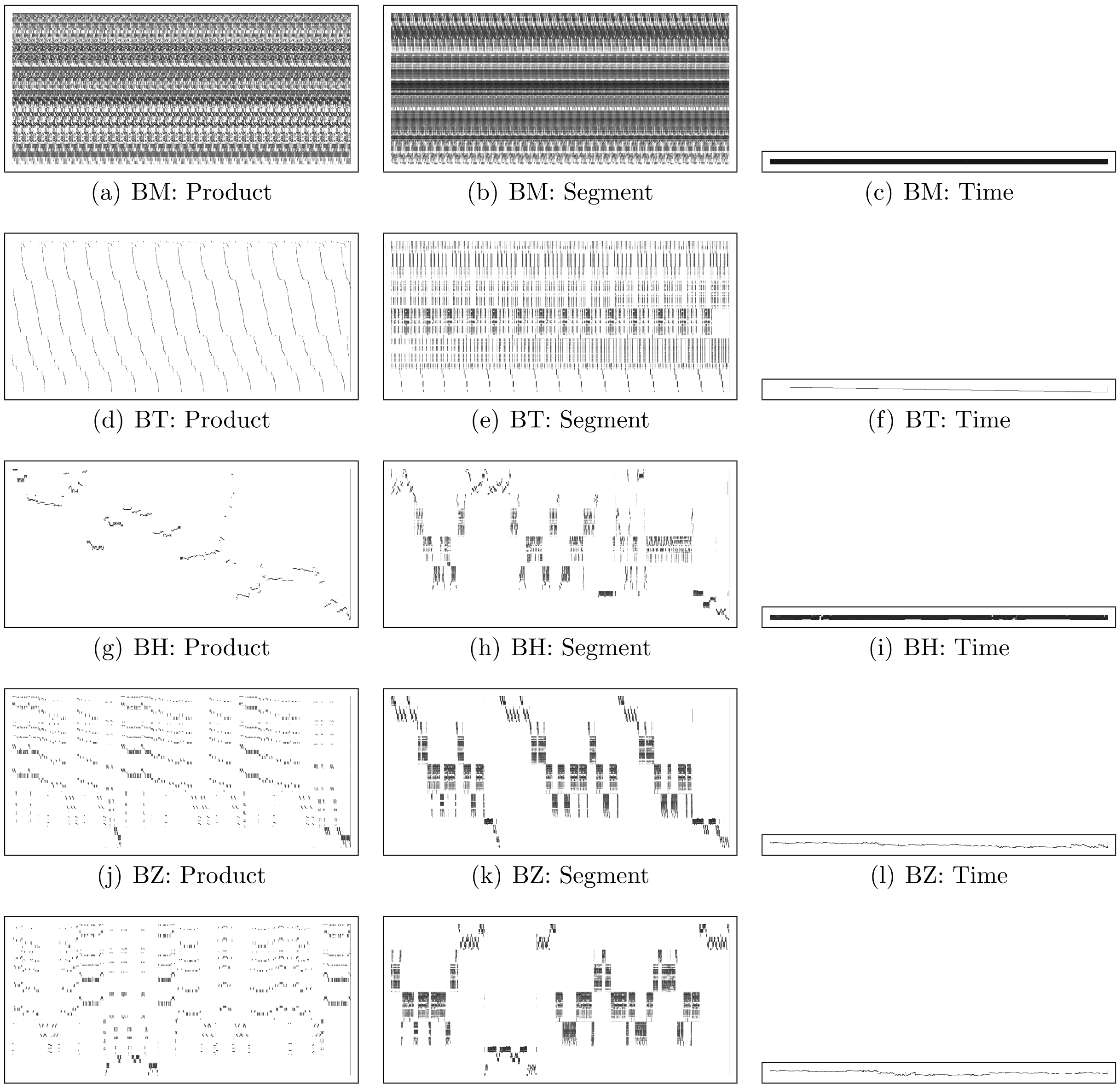
There is a relation between the time required to create a bitmap index and the resulting size of the bitmap. All bitmap indexes require to read the whole fact table, so the only differences stem from the effort to compress and manage them and the I/O caused by writing the compressed bitmap. During creation we do not have to materialize the whole bitmap in memory, as the already created and compressed parts can be written to disk. Only the uncompressed tails of all existing bitmaps need to be kept in main memory. For each inserted tuple we append to the bitmaps corresponding to its attribute values.

“0”-runs between “1”s are generated on the fly as follows. The position of the last appended “1” p_l is saved when appending to a bitmap. The next time a “1” gets appended a “0”-run with length $p_n - p_l - 1$ is appended before it when $p_n - p_l > 1$ holds.

Clustering increases the average run length by storing tuples near to each other on disk, i.e., neighboring tuples have a high probability to have at least some attribute values in common. Thus clustering leads to significantly smaller bitmaps by increasing the run-lengths, especially of the “0”-runs. Further, fewer lookups are necessary for locating the affected bitmaps when attribute values are not changing so frequently.

BM has no clustering at all resulting in the longest creation time and the largest bitmaps. When comparing the *Product* and *Segment* bitmap indexes of BM we see that although they are nearly equal in size creation time of *Product* is five times longer. This is caused by the higher cardinality of *Product* which requires to maintain more bitmaps and causes a much higher load due to more frequent search and switching to the affected bitmap. The bitmap index on *Segment* is larger due to the lower cardinality of *Segment* causing more 1-bits per bitmap and consequently shorter runs which deteriorates the compression.

As expected Hilbert clustering performs best w.r. to overall creation-time and size, as it provides the best clustering. Although, when comparing the relation between the bitmaps



(a) BM: Product (b) BM: Segment (c) BM: Time

(d) BT: Product (e) BT: Segment (f) BT: Time

(g) BH: Product (h) BH: Segment (i) BH: Time

(j) BZ: Product (k) BZ: Segment (l) BZ: Time

(m) BH2: Product (n) BH2: Segment (o) BH2: Time

The x-axis denotes the tuple position, the y-axis the bitmap for an associated value. The origin is in the top-left corner. The darker a pixel the more bits have been mapped to it. We select the brightness by a logarithmic scale as otherwise virtually all pixels would be black. A black pixel represents the maximum number of bits mapped to a pixels for this image. A white pixel means no bits have been mapped to it.

Figure 5.5: Bitmaps mapped to Images

on each dimension it differs from the other SFC-clustered bitmaps. This is a direct result of the fact that it prefers *Product* during clustering as this dimension is the one with the largest cardinality, i.e., differences in higher order bits of the address come all from the *Product* dimensions. Consequently the clustering of the other dimensions is deteriorated and thus those bitmaps are larger than those of the other SFC-clustering bitmap-indexes. See also Figure 3.7 on page 38 for the difference between BH and BH2.

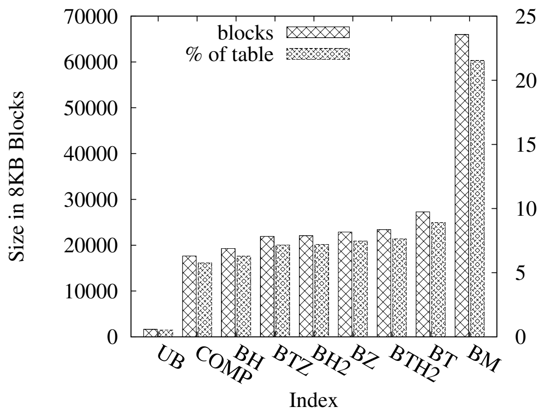
Further, BT shows a high correlation between *Time* and *Product*, i.e., its index on *Product* is the smallest one, even smaller than the one of BH. However *Segment* is just slightly correlated to *Time* and also has shorter runs due to its smaller dimension cardinality anbd thus is nearly as big as for BM.

Figure 5.5 illustrates the resulting bitmaps for some orderings. The bitmaps for one dimension have been mapped to an image by projecting the tuple position to the x-axis and the bitmaps to the y-axis. The attribute values corresponding to the bitmaps have been normalized. The origin is in the top-left corner, i.e., first pixel in the top-left corner corresponds to the first bit in the bitmap with the smallest attribute value. There is only a single 1-bit per row in all bitmaps for a dimension, but on the images there are in fact more pixels per row. This is caused by scaling the domains of attribute value and row positions to the size of the image and thus mapping multiple bits to a single pixel. The image size is 1024x457 pixels for the *Product* and *Segment* dimension and 1024x15 pixels for the *Time* dimension. With 42M rows in the fact table there are 41863 rows mapped to a single pixel on the x-axis. For *Product* 789 bitmaps have been mapped to one pixel in the y-axis and for *Segment* 21 bitmaps.

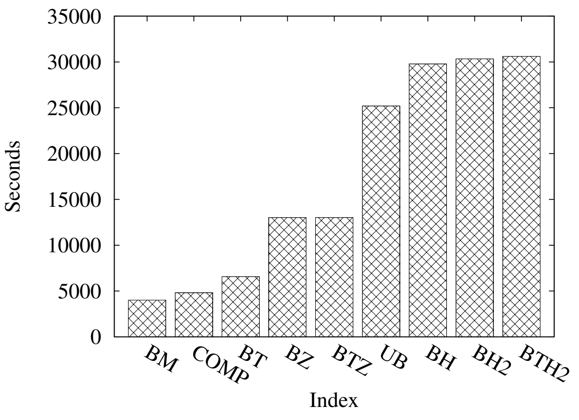
The sizes of the bitmaps in Figure 5.4(b) on page 107 are directly related to the fuzz in the corresponding image, i.e., BM is very fuzzy for all dimensions, see Figure 5.5(a), 5.5(b), and 5.5(c). BT shows a line for *Time* from the origin to the bottom-right corner in Figure 5.5(f) and a correlation of the dimensions *Product* and *Segment* to *Time* in Figure 5.5(d) and 5.5(e). Also the superior compression of BH for *Product* is directly visible in Figure 5.5(g), i.e., the 0-runs are the longest ones resulting in image-lines which are empty except for a few pixels in a local range. BZ and BH2 have similar sizes but different images, while BZ has more noise in the *Product* dimension BH2 has more in the *Time* dimension. The quad-tree partition underlying both curves is also visible in similar patterns which differ w.r. to their location due to the different order in which sub cubes are visited by the curves.

5.3.1.3 Total Index Sizes

Figure 5.6(a) depicts the total index sizes and its relation to the table size. BM, the biggest index, requires 22% of the table size, the other bitmap indexes have approximately 6.6%, COMP 5.9% and UB just 0.54%. Clustering also clearly shows its advantages w.r. to the resulting index size, where BH is the overall winner as already seen in the last section. UB has the smallest size as it just indexes data pages, but not each tuple itself like the bitmap indexes and the COMP.



(a) Total Index Sizes



(b) Total cost for maintenance

Figure 5.6: Overall Maintenance Cost

5.3.1.4 Total Maintenance Time

The sum of all operations required to load the data is depicted in Figure 5.6(b). The main differences stem from sorting data according to the desired order.

BM is the fastest here as it does not require any sorting. It is followed by COMP requiring sorting according to the compound order. UB has a higher cost due to its implementation which does not utilize the SQL loader, but performs insert statements. With a kernel implementation one can expect loading time for UB similar to COMP. Then the Z- and Hilbert-curve based orderings follow.

While clustered indexes are the clear winner w.r. to size this does not come for free, but requires substantially more computation time before loading. It takes 4.5 times longer to gain a size reduction of 30% from BT to BH. Thus this extra effort should pay off during query processing, which will be inspected in the next chapter.

5.3.2 Query Performance

While clustering creates a significant extra effort during loading it should show a benefit during range query processing by maintaining spatial proximity.

The query performance has been measured with two sets of real world queries as they were actually used by the GfK. The largest part of the delivered reports is restricted to a single product group, a single country, and a single two-month period. We label this class of queries *PG_SERIES*. Another important class of queries are running reports restricting time to a range of two consecutive years. We label this class *2Y_PG_SERIES*. Additionally, we performed a *cube query* restricting each dimension to a given percentage of its domain in order to see dependencies w.r. to the result set size.

The resulting restrictions w.r. to their type and selectivity for the 3d universe of GfK are given in Table 5.4 on page 112.

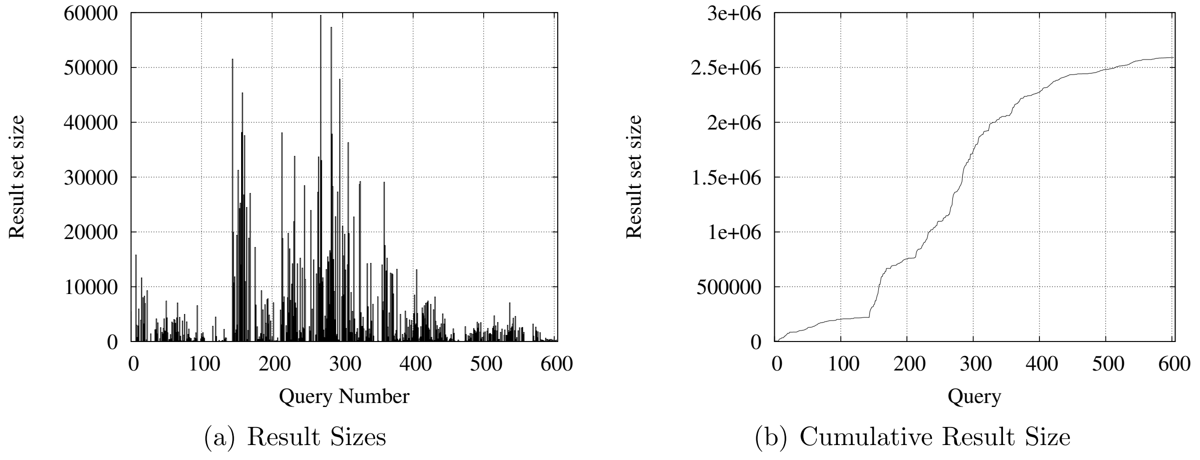


Figure 5.7: *PG_SERIES*: Result Size

Each measurement series was started with a cold cache, i.e., we performed a database shutdown and startup before and flushed disk caches. Within a series the caches were not flushed as this also would not happen when processing the reports.

Oracle was configured to choose its execution plan by its rule based optimizer, which for the bitmap indexes always resulted in the plans listed in Table 5.5 and Table 5.6. The plans directly reflect the restrictions types. A range restriction causes a range scan to determine the bitmaps for those values in the range and merges them. A single value restriction does not require a merge (line 4 of Table 5.5), but the resulting bitmap directly goes into the `AND`ing. The bitmaps resulting from the dimension restrictions are `AND`ed and converted to ROW-IDs before accessing the fact table.

COMP was always processed by a range scan, as listed in Table 5.7, i.e., the first dimension in the indexed key determines the range. For these measurements it was *Time*.

All series were run twice, once to get *execution times* and a second time with profiling enabled in order to obtain the *number of accessed pages* and execution plans.

The queries within a series were sorted in the same order as for COMP. This favors COMP due to better cache utilization, but it is also the natural order as the results are grouped for the customer reports, thus we used this query-ordering for all indexes.

The measured metrics only include fact table accesses, but they do not include any further operations (sorting, joins with the dimension tables, aggregation, etc.) as these are the same for all bitmap indexes.

For enhanced aggregation algorithms utilizing the Z-curve- and MHC-clustering we refer to [Zir03] and [Pie03].

5.3.2.1 *PG_SERIES*

Result Set Sizes: The result set size and the cumulated size are plotted in Figure 5.7. The first 144 queries have small result sets, i.e., 0 to few thousand tuples, then larger

	PG_SERIES		***2Y_PG_SERIES***		**CUBE with 5% steps**	
	Type	Selectivity	Type	Selectivity	Type	Selectivity
Product	Range	$\frac{1}{604}$	Range	$\frac{1}{604}$	Range	$[0 : x\%]$
Segment	Range	$\frac{1}{16}$	Range	$\frac{1}{16}$	Range	$[0 : x\%]$
Time	Point	$\frac{1}{15}$	Range	$\frac{6}{15}$	Range	$[0 : x\%]$
Total		0.0006%		0.004%		0.000125 . . . 12.5%
# Queries	604		604		10	

Table 5.4: Query Sets for GfK DWH

```
TABLE ACCESS (BY INDEX ROWID) OF 'GFKB'
   BITMAP CONVERSION (TO ROWIDS)
      BITMAP AND
         BITMAP INDEX (SINGLE VALUE) OF 'GFKB_TIME_CS'
         BITMAP MERGE
            BITMAP INDEX (RANGE SCAN) OF 'GFKB_PRODUCT_CS'
         BITMAP MERGE
            BITMAP INDEX (RANGE SCAN) OF 'GFKB_SEGMENT_CS'
```

Table 5.5: *PG_SERIES* Execution Plan for GfK DWH

```
TABLE ACCESS (BY INDEX ROWID) OF 'GFKB'
   BITMAP CONVERSION (TO ROWIDS)
      BITMAP AND
         BITMAP MERGE
            BITMAP INDEX (RANGE SCAN) OF 'GFKB_PRODUCT_CS'
         BITMAP MERGE
            BITMAP INDEX (RANGE SCAN) OF 'GFKB_SEGMENT_CS'
         BITMAP MERGE
            BITMAP INDEX (RANGE SCAN) OF 'GFKB_TIME_CS'
```

Table 5.6: *2Y_PG_SERIES* and Cube Execution Plan for GfK DWH

```
      INDEX (RANGE SCAN) OF 'GFKCOMP' (UNIQUE)
```

Table 5.7: COMP Execution Plan for GfK DWH

results follow and finally results get smaller again. We have chosen cumulative plots where appropriate as they are smoothing the plots. Otherwise, most plots would be so fuzzy that the different indexes cannot be distinguished anymore.

Elapsed time: The elapsed time is the most important measure for users. It inherently includes I/O waits and CPU load. Our measurements have been performed in a single user environment and thus should not be transfered to multi-user environments without also taking the physical and logical page accesses into account as those will provide more insight into the overall performance.

Figure 5.8 on the next page shows the elapsed time. The labels of the curves are in the same order as the curves last position, i.e., BM is the topmost curve and UB the bottommost curve in the cumulated plot Figure 5.8(a). BM requires 21 times longer than UB and 5 times longer than BH, thus we have restricted the plot to the maximum of BH in order to better see the differences.

UB performs best, being about twice as fast as BH2, which is the best bitmap index. The better embedding of the universe in BH2 and BTH2 shows a clear advantage over BH, resulting in less than half of the execution time. BH works well with small results sets, but as results are getting bigger (after query 237; arrow 1 in Figure 5.8(a)) the bad clustering leads to more cache misses and thus more physical page accesses resulting in longer execution times.

COMP lies within BTH2 and BH w.r. to the overall time, but for small results (the first 144 queries) in the beginning it does not perform as good, since it is not able to utilize restrictions on all dimensions.

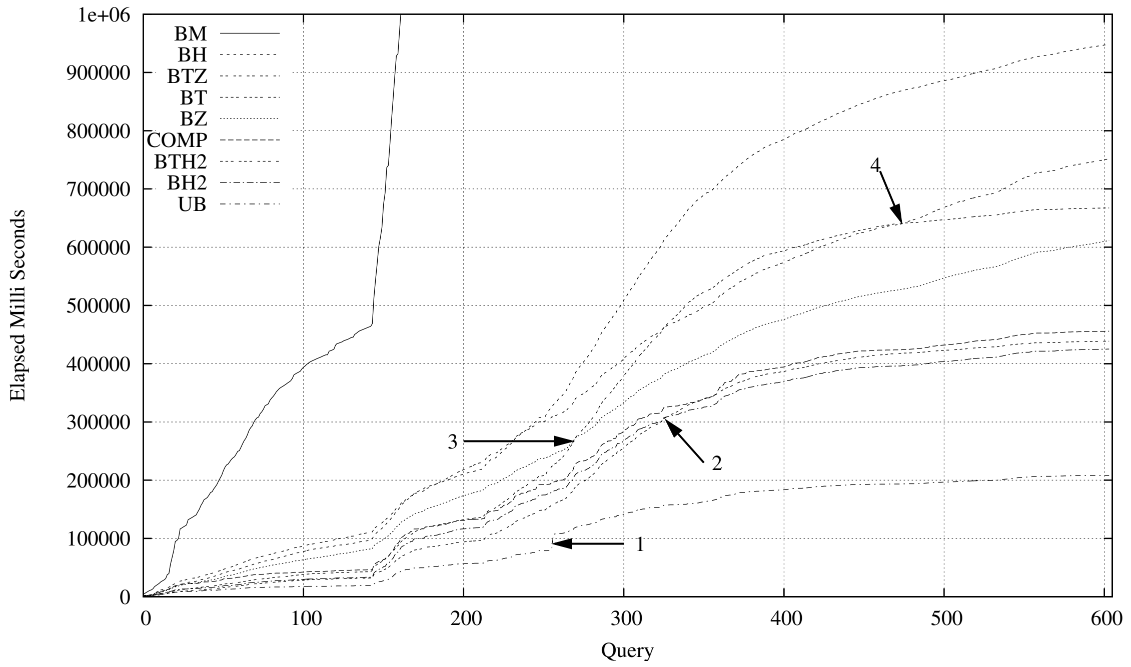
BT performs very well for small result sets due to the correlation of *Product* and *Time* those do not cause too much random accesses. For small results it even performs better than BZ. However, it degenerates with increasing result sets, as can be seen for queries 145 to 400, which are correlated to the result set size. The increase becomes steeper with larger result sets (arrow 3 in Figure 5.8(a)) and declines again with smaller result sets arrow 4 in Figure 5.8(a).

For the BTH2 and BTZ orderings we see a decrease of the execution times as their clustering is not as optimal as for BH2 (arrow 2 in Figure 5.8(a)) resp. BZ. This influence is more significant for the Z-curve than the Hilbert-curve.

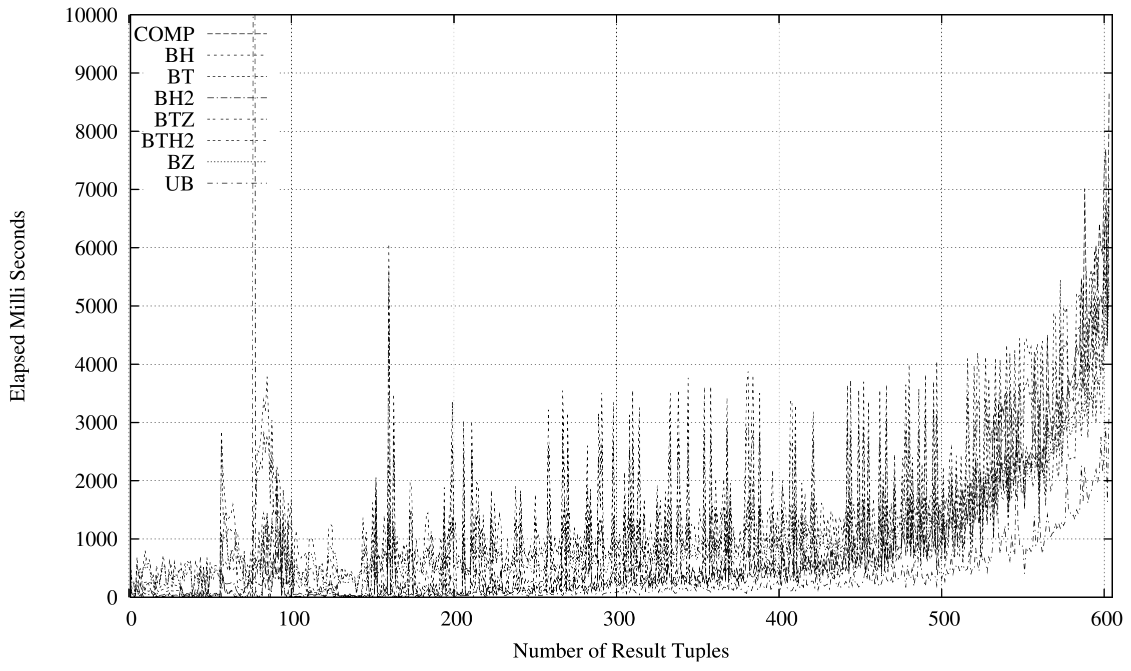
Figure 5.8(b) shows the times w.r. to result set size. The times are correlated to the result set size, but for the bitmap indexes there is a high variation in execution times. We have omitted BM from this plot as it is too fuzzy and steep. UB performs here also best with the fewest fuzz, while BH and BT have the highest fuzz.

Page accesses: Logical page accesses (lRs = logical Reads) reflect the number of all accessed pages. Logically accessed pages have to be processed and thus cause CPU load. A lR is a physical page access (pRs = physical Reads) if the required page is not located in the cache and thus has to be fetched from disk causing I/O. Physical page accesses indicate how I/O bound and cache sensitive an index/clustering is.

An index minimizing both, increases the overall performance. With $pR = 0$ for a query



(a) Cumulative Time



(b) Time

Figure 5.8: *PG_SERIES*: Elapsed Time

there is no I/O, thus the query is totally CPU bound by lRs. With $pR = lR$ the query is I/O bound, unless page processing is more expensive than fetching the page.

All the page accesses presented here have been measured by tracing the execution of queries with the Oracle profiler. Thus, we cannot distinguish between data and index pages and consequently the presented numbers include both.

Page accesses w.r. to query order: Figure 5.9 on the following page shows pRs and lRs with a logarithmic scale on the y-axis. The cumulative plot is better suited to visualize the overall differences, since the number of accessed pages has a high variation for bitmap indexes. In the beginning there is a steep increase for all indexes until caching shows its effects.

As expected BM performs worst, followed by COMP and BH. Then all the other clustering bitmap indexes follow and finally UB with the best performance requiring just $\frac{1}{100}$ to the pRs of BM.

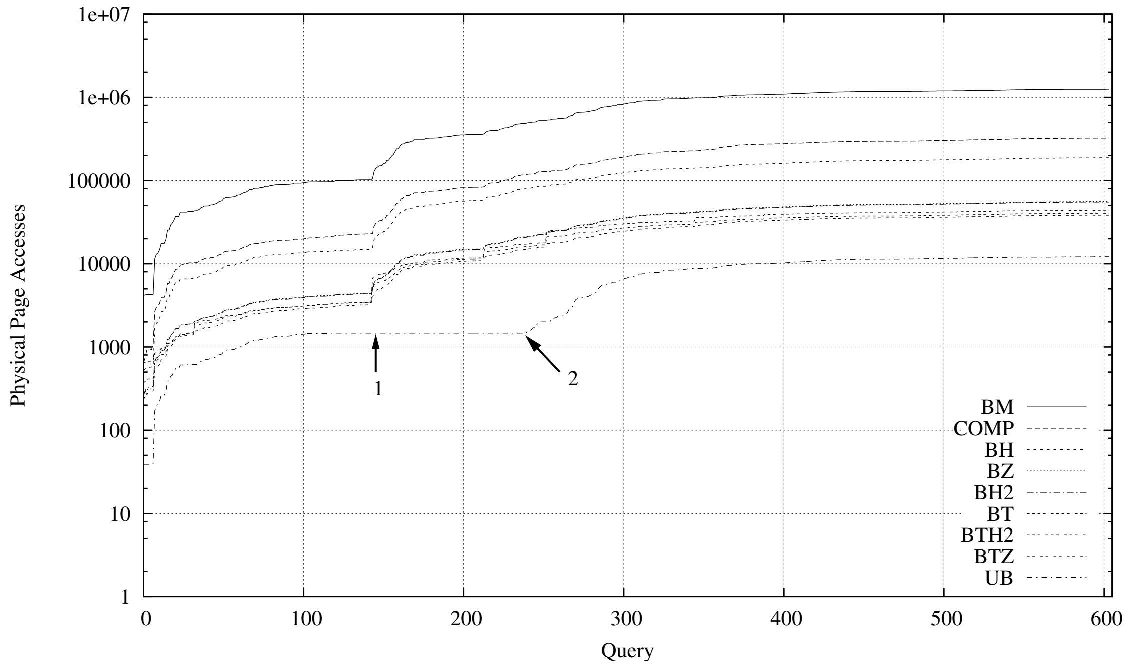
UB has no increase after query 144 in the physical page accesses (arrow 1 in Figure 5.9(a)), but stays steady up to query 238 (arrow 2 in Figure 5.9(a)). Due to its smaller index the cache can be utilized for more data pages causing no I/O where other indexes require it. But, the bigger result sizes are still visible in the elapsed time and the logical page accesses showing the jump after query 144.

The pRs for BZ and BH2 are essentially equal. Also the lR are the same thus BH2 has a higher locality for the pRs causing its better performance.

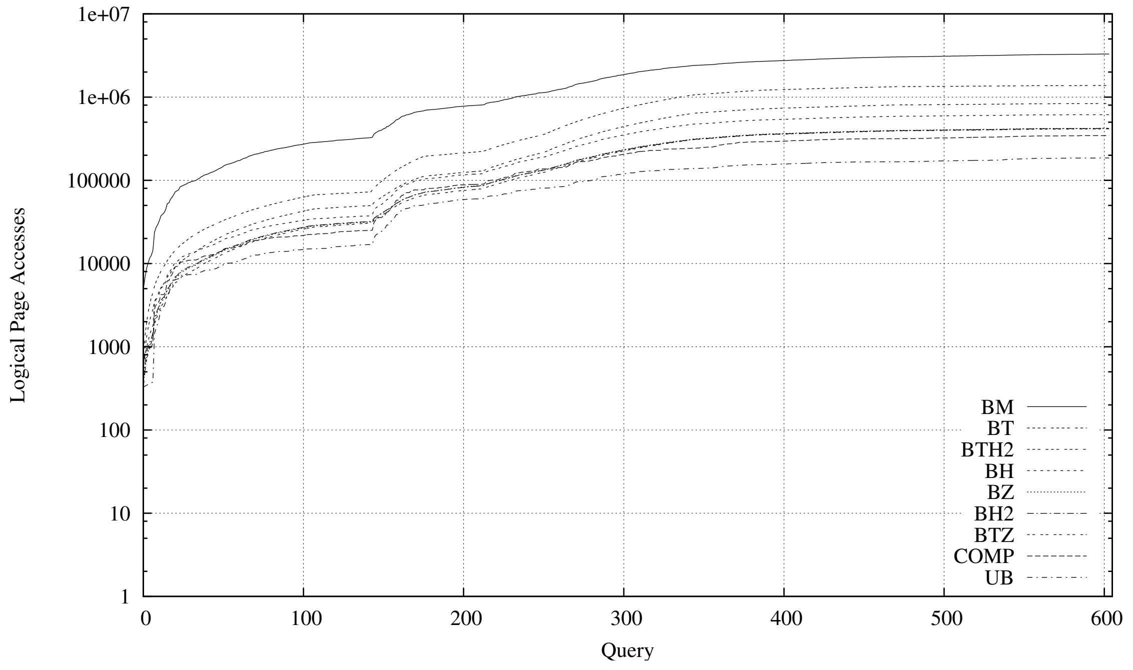
While COMP performs quite well w.r. to time it does not when it comes to the accessed pages. The number of logical page accesses is nearly the same as the number of physical accesses while for other indexes they are substantially higher causing a higher CPU load and longer times. Thus although COMP requires more physical page accesses it benefits from having fewer logical ones. COMP performs bad for big results sets, but good for very small or empty result sets and thus is getting good results w.r. to the over all time.

Another interesting observation from the measurements is the difference of the bitmap indexes with time as the first sort key to those that do not have it, i.e., BZ to BTZ and BH2 to BTH2. In general having time as a prefix pays off w.r. to the physical accesses. Unfortunately, BTH2 has more logical accesses, while being higher they must be more local than for BH2 otherwise the better performance of BTH2 is not explainable. Although BTZ has better physical and logical access numbers its performance w.r. to time is not as good as BZ. Essentially BZ seems to have a higher locality, i.e., while accessing more pages it is cheaper to access them as they are better clustered.

Page accesses in relation to each other and time: Figure 5.10(a) shows the relation of physical page accesses to logical page accesses. The major cost of BT stems from logical accesses. Also it was already noted, that COMP is doing well w.r. to elapsed time. Although, COMP has the second most physical accesses, is has much fewer logical accesses, i.e., 90% of its logical accesses are physical ones.

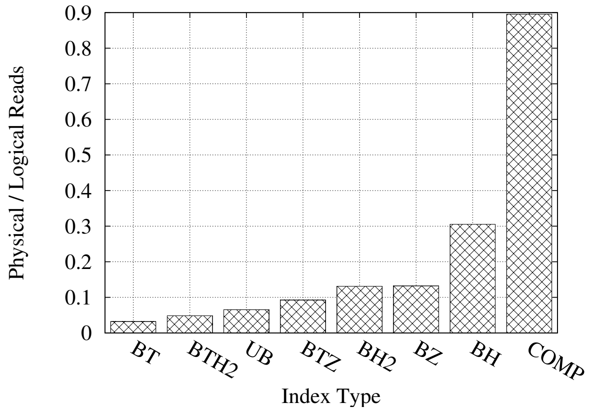


(a) Cumulative Physical Page Accesses

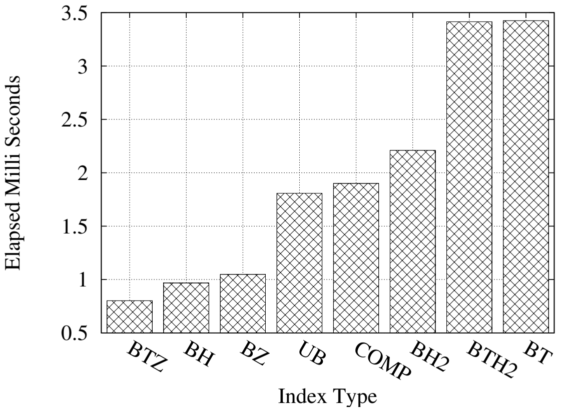


(b) Cumulative Logical Page Accesses

Figure 5.9: *PG_SERIES*: Page Accesses w.r. to Query



(a) Physical / Logical Page Accesses



(b) Time per Page Accesses served from Cache

Figure 5.10: Time per Logical Page Accesses

Figure 5.10(b) depicts the overall time in relation to the sum of logical accesses. Although, UB performs best w.r. to elapsed time and page accesses, it performs medium here. This is mainly caused by the page processing overhead of the implementation, i.e., the UBAPI which is implemented as client-side application.

Page accesses w.r. to result set size: Figure 5.11(a) shows the physical page accesses w.r. to the result set size.

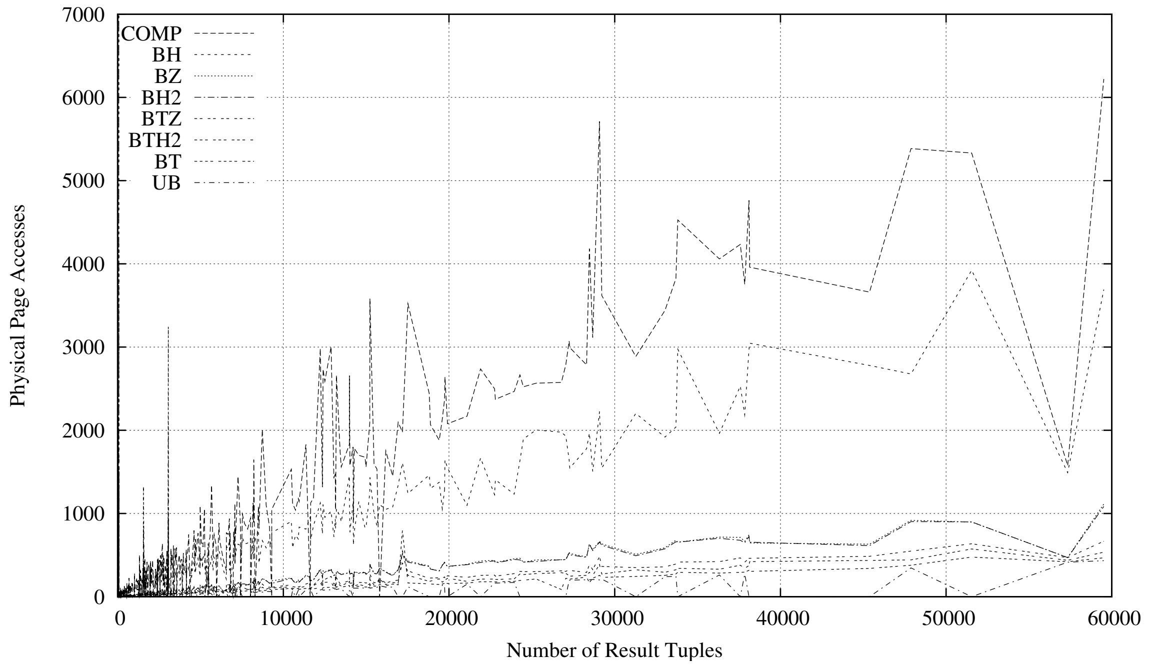
The clustering bitmap indexes and UB show a correlation to the result set size. UB has the fewest jumps. This predictable dependency is a big advantage for query optimization, as it allows for more exact estimations of execution times and thus allows for choosing the better access path.

BT has some extreme outliers in Figure 5.11(a) two peaks are visible in the beginning followed by others. Those peaks stem from the first queries on a new time period accessing a huge amount of pages which are then in cache for subsequent queries on this period. The remaining queries are mainly served from cache but still cause logical accesses and make BT having the second most logical accesses in Figure 5.9(b).

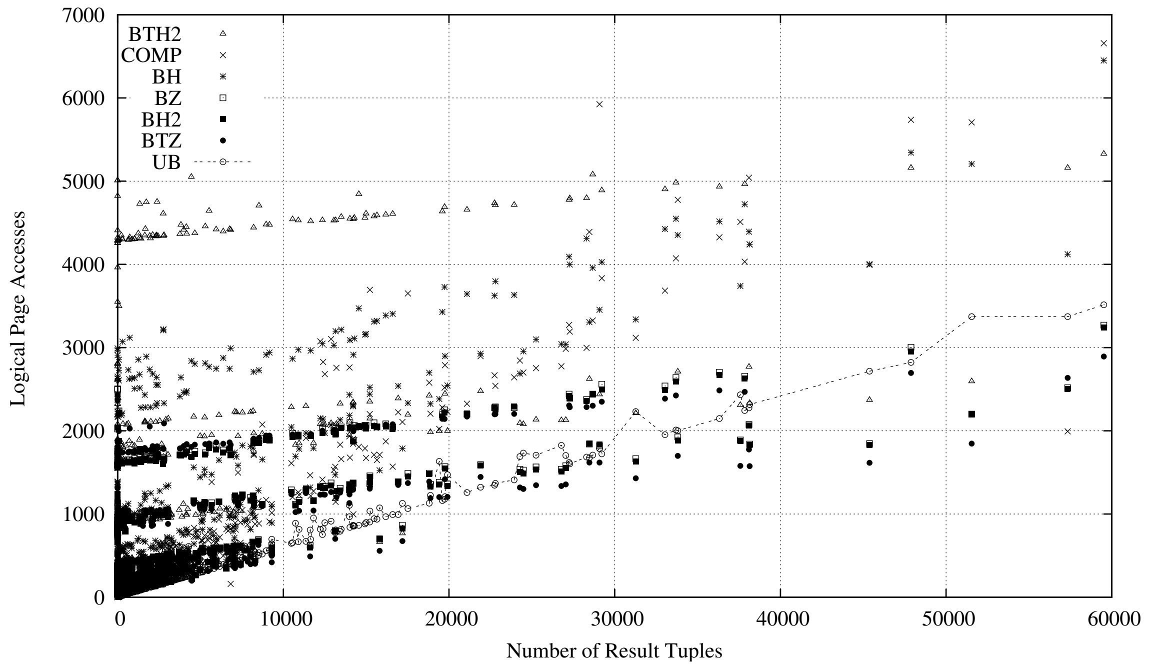
Figure 5.11(b) shows the logical page accesses w.r. to the result set size. The fuzz is here much higher than for the physical accesses thus the actual effect of clustering on the page accesses is alleviated by caching.

For result sizes greater than 25000 tuples UB is sometimes outperformed by bitmap indexes. This is caused by better clustering, i.e., SFC-region boundaries fit better to the specific query causing fewer intersections with regions.

Another interesting observation is the effect of clustering with a time prefix, i.e., BTH2 and BTZ. The Hilbert clustering is clearly deteriorated by adding time as a prefix to the SFC-address. There are many outliers of BTH2 on the top of the plots, i.e., the number of logical accesses is hardly predictable. On the other hand, BTZ benefits from adding the time prefix performing better than BZ. An explanation for both of these phenomena can be

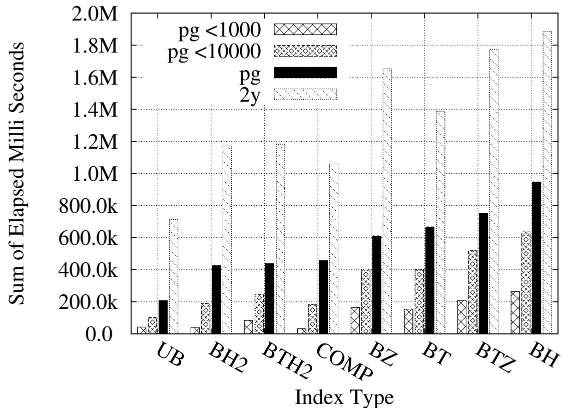


(a) Physical Page Accesses

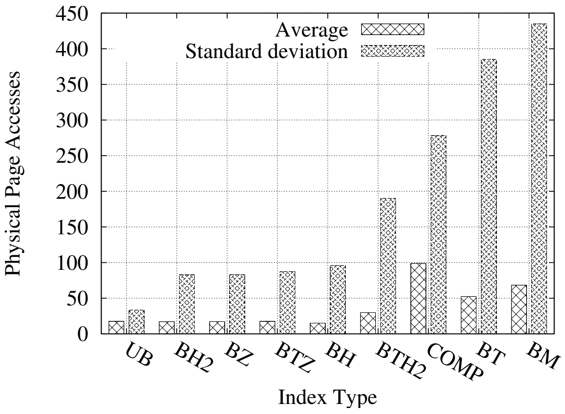


(b) Logical Page Accesses

Figure 5.11: *PG_SERIES*: Page Accesses w.r. to Result Set Size



(a) Sum of Elapse Times



(b) pRs for empty result sets

Figure 5.12: *PG_SERIES* Summary

found in the correlation of time and product and the specific properties of the SFC-curves. The Z-curve preserves the dimension order in the SFC-address while the Hilbert-curve does not. This sometimes causes points to be farer away[2] on the SFC for the Hilbert-curve than for the Z-curve.

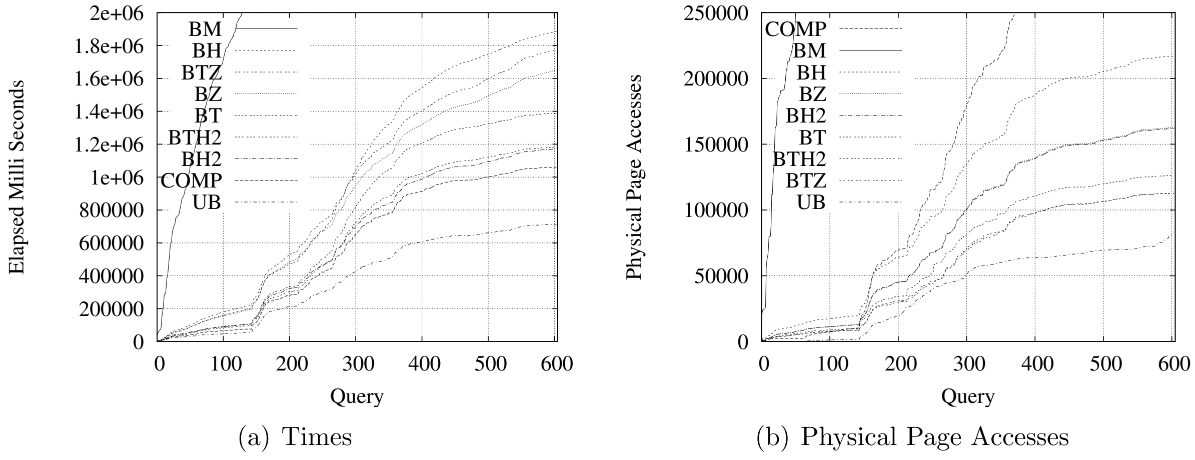
Result Set Sizes: Figure 5.12(a) shows the sum of elapsed times for the *PG_SERIES* with result sets smaller than 1000 resp. 10000 tuples and for all queries for *PG_SERIES* and *2Y_PG_SERIES*. UB also wins here, only COMP is better for the small result sets due to being integrated into the DBMS kernel and finding empty or small result sets quickly.

Figure 5.12(b) shows the average and the standard deviation for those 1004 queries with empty results. While UB is not the one with the smallest average it is the one with the smallest standard deviation. While the bitmap indexes do not require to access data pages they require to read all related bitmaps. BH with the smallest bitmaps in sum is here slightly better than UB. BT and BM with the biggest bitmaps have the highest deviation. Also COMP has a higher average and deviation, since if the restriction on the first attribute is not empty it needs to scan the range for testing the other restrictions.

Summary: There is one clear winner w.r. to query performance: The UB-Tree is the fastest index with the best cache utilization and a nearly linear dependency of elapsed time w.r. to logical page accesses. With a UB-Tree integrated into the kernel there would be further decrease of execution times by a factor of ≈ 4 [RMF+00].

Further, the measurements show a significant advantage when clustering data indexed by bitmap index. The Hilbert based clusterings have $\approx 30\%$ shorter execution times than BZ, which is caused by the better clustering, i.e., fewer queried segments on the SFC. However, the number of page accesses is not significantly different between those two

[2]See also the distribution and maximums for the Hilbert- and Z-curves in Section 3.4.1



(a) Times (b) Physical Page Accesses

Figure 5.13: *2Y_PG_SERIES*

orderings.

The measurements also clearly show the impact of correctly embedding universes with differing dimension cardinalities for the Hilbert curve (BH vs. BH2).

The SFC-clustered bitmap indexes also provide a significant performance improvement over a clustering by time (BT), especially w.r.t. to elapsed time and logical accesses.

The theoretical results from Chapter 3 are supported by these measurements.

5.3.2.2 *2Y_PG_SERIES*

The *2Y_PG_SERIES* have the same restrictions on *Segment* and *Product* as the *PG_SERIES*, but a range restriction of two years instead of a single period.

Figure 5.12(a) on the page before shows the overall difference in comparison to the *PG_SERIES*. The overall result size of *2Y_PG_SERIES* is nearly 8 times larger than for *PG_SERIES*, but the elapsed times are only ≈2-3 times higher as pre-fetching and caching show their positive effects here.

COMP, BM and BT have an ≈50% increase in elapsed time, while the clustering indexes have a higher increase as their advantages weakened by pre-fetching and caching.

The results of the measurements are qualitatively the same for the execution times as depicted in Figure 5.13(a), but are higher due to the larger result sizes. UB again performs best w.r. to elapsed time, COMP performs second best followed by BH2 and BTH2. Then BT, BZ, BTZ, and BH follow.

Figure 5.13(b) shows the number of physical page accesses. Due to the bigger range in *Time* there are no differences between Z-curve and Hilbert-curve based bitmaps anymore, but they perform equally well: BZ as good as BH2 and and BTZ as good as BTH2. While this only shows the number of accessed pages it does not show the quality of the clustering w.r. to fewer segments on the curve and a higher locality of the accessed pages for queries. The Hilbert-curve based mappings are better here and thus also perform better w.r. to elapsed time.

UB again performs best w.r. to all measures. While being about twice as fast as the best bitmap index for the *PG_SERIES*, it is only ≈40% better for the *2Y_PG_SERIES* due to the page processing overhead of the UBAPI.

5.4 Summary

This chapter has discussed the effects of clustering a fact table by a SFC which is indexed by a bitmap index. After introducing the concepts of bitmap indexes we have presented comprehensive measurements confirming the theoretical results from the previous sections (Chapter 3 and this chapter). We have also added measurements with the UB-Tree and a compound index in order to see how they perform in comparison to bitmap indexes.

In the following we list the main results:

- Clustering significantly reduces the size of bitmap indexes in comparison to no clustering. However, the differences between the clusterings are below 3% in comparison to the table size. (Figure 5.6(a) on page 110).

- Address calculation for the Hilbert-curve requires ten times longer than for the Z-curve. Please note the following:

  Address Calculation: as far as we know, we have used the best implementation of the algorithm [Moo00] for the Hilbert-curve address calculation which is still flexible to support an arbitrary number of dimensions. The Z-curve address calculation is a subpart of the Hilbert-curve address calculation.

  Query Processing: we have discussed that the NJI/NJO calculation is very costly for the Hilbert-curve (Section 3.2.1.2 on page 44). However, as these operations are not necessary for an access by bitmap indexes they are not part of the results in this chapter.

  Correct Embedding: the results clearly show that the query performance for the Hilbert-curve is deteriorated when embedding the dimensions into the quadratic Hilbert universe in the wrong way.

- The used compound indexes is a well performing alternative in comparison to bitmap indexes for this application. They are causing more physical accesses than the other indexes but this is compensated by fewer logical page accesses.

- The UB-Tree (as UBAPI) outperforms all other indexes w.r. to all metrics. Its index is nearly twelve times smaller than the best bitmap index. It performs two times faster than the best bitmap index. With an integrated version of the UB-Tree further improvements are possible, especially loading should improve being as good as for COMP.

The CUBE measurements have not brought any additional insights and thus are not discussed in an own section. All clustering bitmap indexes scale here linear to the result set size. UB degenerates with bigger result sets due to the used implementation, i.e., the UBAPI.

5.5 Multi-User environments

Our measurements have been conducted in a single user environment in order to avoid any side-effects of other processes. Therefore, the elapsed times as measured here cannot be directly transfered to multi user environments. In multi user environments the impact of logical page accesses is much higher as the CPU is not exclusively used for one query. Z-curve based orderings have been better or equal w.r. to logical accesses (Figure 5.11(b) on page 118) than Hilbert-curve orderings and thus we expect them to perform better here, especially we observed that the processing of queries by bitmap indexes showed a 99% CPU load most of the time. With a kernel integrated version of the UB-Tree, also the cost per logical page (Figure 5.10(b) on page 117) access would significantly decrease.

The UB-Tree also offers clear advantages here as it has the fewest logical and physical page accesses and thus will perform best in multi user environments.

Chapter 6

The BUB-Tree

6.1 Introduction

Real world data is usually not distributed uniformly, i.e., there are clusters of data, but most of the universe is unpopulated. To the unpopulated space we will refer to as *dead space*. Thus with skewed data distributions there are usually large areas of dead space.

It is important that an index detects dead space as soon as possible in order to prune search paths. The performance of the index should not degenerate for queries covering dead space. One can determine if an index handles dead space efficiently by checking the page accesses for empty results. There should be only a few page accesses for such queries, i.e., for tree based indexes not more than h accesses where h is the tree height. Also, when available, a high number of "false" page reads, i.e., pages not contributing any tuples to the result, is an indicator that dead space is not handled well. Further, an index might not handle dead space by design, e.g., the B-Tree and UB-Tree.

The UB-Tree partitions the whole universe as it only stores the separators between regions. Therefore, dead space queries always have to go down to the level of data pages. Only fetching the data pages and inspecting its content ensures that there is no data. While this is no problem for a point search it can quickly become a problem for range queries processed as a set of SFC segments causing h page accesses for each segment.

To solve this problem of the UB-Tree, we propose the bounding UB-Tree (BUB-Tree), a UB-Tree storing the bounds of Z-regions in its index pages. We present modified algorithms for maintenance and how to utilize this additional information for query processing. The BUB-Tree preserves the good properties of the UB-Tree, i.e., logarithmic worst case guarantees for the basic operations of insertion, deletion, and point query. During query processing, the extra information can be used to prune search paths and thus reduce the number of page accesses.

Applications causing queries also covering dead space are:

Collision detection: This is important for navigation systems, especially for fast objects moving freely in space a short response time is important. The database usually will store the allocated positions and a frequent answer to queries is the empty result, i.e., "go ahead there is no danger".

Finding free slots: This is related to collision detection, but the data base stores free positions. If the position of a free slot matters, we want to get a response quickly, either finding a free slot in a designated area or nothing in order to start a new search somewhere else.

Skewed queries:

Partial match queries: restricting only a subset of the indexed attributes results in hyper-plane queries. Depending on the partitioning, this might cause queries cutting the partitioning like a puff-pastry orthogonal to the partitioning.

Dual space queries: This is a mapping of spatial objects to points resulting in skewed data distribution and queries covering large portions of dead space. This will be discussed in more detail in Chapter 7.

Skyline queries: Processing them by a NN-search [KRR02] results either in significantly higher processing cost to avoid duplicates or in query ranges always including the origin. This will tend to generate query shapes similar to partial match queries.

Data Warehousing: Data cubes are usually sparsely populated ($\ll 1\%$) and the data distributions are highly skewed (see Appendix B.1). Utilizing clustering techniques like MHC [Pie03] will further increase this.

6.2 Related Work

Basically, all index structures that are able to prune search paths leading to dead space have been designed for spatial objects and use some kind of object bounding.

The R-Tree [Gut84] and its variants, the R^+-Tree [SRF87] and the R^*-Tree [BKS+90], are the most prominent and accepted approaches to provide good performance. They are able to prune search paths to dead space during query processing, as they are using minimum bounding boxes (MBBs) in their index. However, overlap of the bounding boxes prevents reasonable worst case guarantees, i.e., in the worst case all pages have to be accessed for answering a simple point query. The R^*-Tree tries to minimize the overlap by forced re-insertion, but this tremendously increases the overhead of insertions. Also various bulk loading algorithms have been proposed trying to minimize overlap of bounding boxes, either by partitioning the data, e.g., the STR approach [LEL97], or by sorting it according to a SFC, e.g., the Hilbert-curve [KF93].

Other index structures are the P-Tree [Jag90b, Sch93] using polygons for bounding objects, the SKD-Tree [Ooi87, Ooi90] storing upper and lower bounds and the GDB-Tree [OS90] storing minimum bounding rectangles.

In contrast to all of these, the BUB-Tree provides the same worst case guarantees as the B-Tree and UB-Tree, i.e., basic operations are logarithmically bound by the height of the tree. Additionally to the UB-Tree, the BUB-Tree is also able to prune search paths. Further, it is easier to integrate into a DBMS kernel. The only remaining difference is: it is only designed for point data, but not for spatial objects. However, also spatial objects can be handled efficiently with the UB-Tree and the BUB-Tree as discussed in Chapter 7.

6.3 BUB-Tree regions

A tuple in the index part of a (U)B$^+$-Tree consist of a separator and a link to the associated child node. Due to this, the whole data space is partitioned into disjoint, but consecutive intervals on the SFC. Whether the resulting regions cover dead space or not is not known at the index level of the tree.

The fundamental idea of the BUB-Tree is to store two addresses bounding the first and last tuple on the indexed child node w.r. to address order. This still guarantees a disjunct partitioning and thus the worst case performance for the basic operations is still logarithmic as for the UB-Tree. Further, by bounding the populated space, search paths can be pruned during query processing.

Formally, a BUB-Tree can be defined as follows:

Definition 6.1 (BUB-Tree Structure)
A BUB-Tree is a set of disjoint segments $\mathbb{B} = \{[\sigma_1, \epsilon_1], [\sigma_2, \epsilon_2], \ldots, [\sigma_k, \epsilon_k]\}$ on a SFC partitioning the universe Ω where $\epsilon_i \leq \sigma_{i+1} \forall i \in [1, k-1]$.
A B$^+$-Tree on the compound address $\sigma_i \circ \epsilon_i$ indexes the segments. ◇

It is not necessary that σ_i and ϵ_i correspond to the first and last tuple in page i, however when optimally bounding the populated space they will correspond to these. Using the compound key $\sigma_i \circ \epsilon_i$ for the indexing B$^+$-Tree allows to reuse the tree structure without modifying it. Index pages now can only store half the entries of a UB-Tree, but utilizing prefix compression instead of storing the plain keys allows to significantly reduce the overhead. Compressing should use the prefixes of the addresses σ_i and ϵ_i for best results, but not the prefix of the compound key $\sigma_i \circ \epsilon_i$.

Pushing the concept of intervals further up into the index will result in index entries bounding all child nodes in the node they refer to. This is similar to the hierarchy of bounding boxes in a R*-Tree.

As well as for the R*-Tree, also the BUB-Tree allows a page capacity below 50% in order to find a better partitioning of the universe. We will refer to this lower bound for the page utilization as U_{min}. While it is a property of the BUB-Tree, it can be overwritten by a new U_{min} during insertion or reorganization.

Example 6.1 (Structure of a BUB-Tree)
Figure 6.1 on the following page shows nine data points in relation to their position on the SFC. The points are grouped into data pages with a capacity of two tuples.

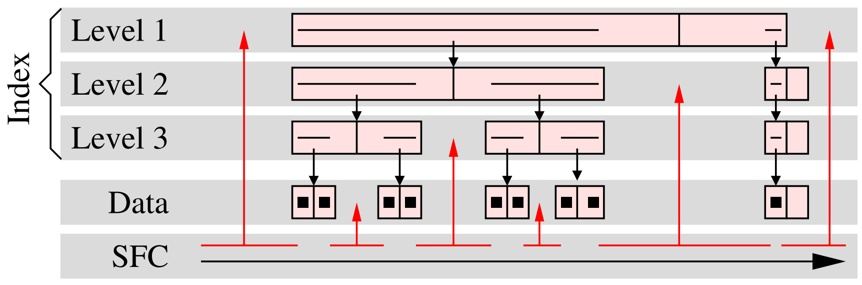


Figure 6.1: Data, SFC-segments, and Pages of a BUB-Tree

The intervals as stored in the BUB-Tree are displayed according to the index levels 1 to 3 where 1 corresponds to the level of the root. Index pages have a capacity of two index entries. The hierarchy of indexed SFC-segments is visible by top-down arrows from the corresponding pages. The dead space segments not covered by the index are depicted by segment with bottom-up arrows from the SFC to the level of the index where dead space is detected during search. ◇

Example 6.2 (Partitioning of a UB-Tree, BUB-Tree, and R*-Tree)
Figure 6.2 on the next page shows the partitioning of a two dimensional universe for the UB-Tree, BUB-Tree, and R*-Tree on the last level of the index just before the data pages. Each region of the UB-Tree and BUB-Tree is colored in a different color. The bounding boxes of the R*-Tree are filled grey while the actual box boundary is in a unique color. There are two clusters of data shown in Figure 6.2(a). The UB-Tree has large Z-regions also covering dead space.

In contrast to this the regions of the BUB-Tree and R*-Tree closely approximate the populated area and most of the dead space (white area) is recognized by the index. The R*-Tree shows already some overlapping regions, while the BUB-Tree shows small dead space areas between regions. ◇

In order to estimate the quality of a BUB-Tree partitioning we define the coverage of an index, i.e., the amount of space covered by the index. Coverage in this context is defined as the cardinality of the set of all points $\vec{p} \in \Omega$ which require to inspect data pages in order to answer if a point $\vec{p}$ actually exists. We normalize this value w.r.t. to the size of the universe and thus get a percentage value.

We provide no definitions for the B$^+$-Tree and UB-Tree, as they always cover the complete universe their coverage is always 100%. The coverage of a BUB-Tree is defined as follows:

Definition 6.2 (BUB-Tree-Coverage)
The *coverage* of a BUB-Tree partitioning $\mathbb{B}(\Omega) = \{[\sigma_1, \epsilon_1], [\sigma_2, \epsilon_2], \ldots, [\sigma_k, \epsilon_k]\}$ for a universe Ω is defined as

$$cov(\mathbb{B}(\Omega)) = \frac{\sum_{i=1}^{k} vol([\sigma_i, \epsilon_i])}{|\Omega|} = \frac{\sum_{i=1}^{k} (\epsilon_i - \sigma_i)}{|\Omega|}$$

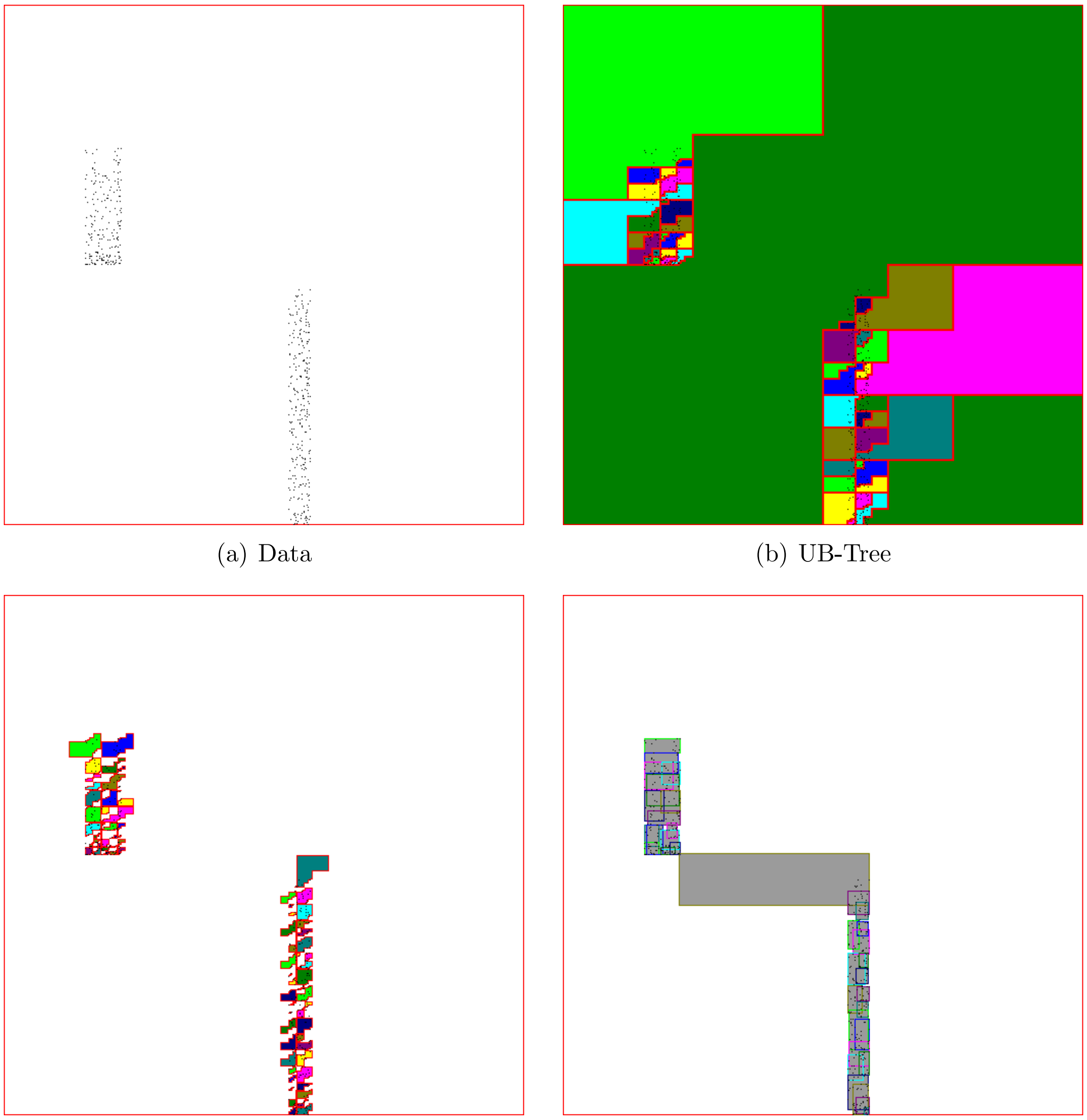

(a) Data (b) UB-Tree (c) BUB-Tree (d) R*-Tree

Figure 6.2: Partitioning of Data resp. Universe for UB-Tree, BUB-Tree and R*-Tree

◇

Definition 6.3 (Optimal Coverage of a BUB-Tree)
A BUB-Tree partitioning $\mathbb{B}_i(\Omega)$ of a universe is called optimal iff, $\nexists \mathbb{B}_j(\Omega) | cov(\mathbb{B}_j(\Omega)) < cov(\mathbb{B}_i(\Omega))$ where $\mathbb{B}_i(\Omega)$ and $\mathbb{B}_j(\Omega)$ have the same size, i.e., the same number of regions. ◇

The goal is to minimize the coverage of a BUB-Tree while retaining a designated page utilization. As there are many possible partitionings for a universe it is not trivial to find the one with the minimal coverage, however we present algorithms trying to minimize the coverage while only causing modifications which are local w.r. to the tree structure of the BUB-Tree. The actual number of possible partitionings is finite as we are indexing a finite set of tuples.

The coverage is also related to the page capacity, i.e., with a capacity of one only points are covered by the data pages and thus the coverage is $cov(\mathbb{B}(\Omega)) = \frac{k}{|\Omega|}$ for a relation with k data pages resp. points.

6.4 Point Query

A B^+-Tree/UB-Tree point search for a point $\vec{p}$ with key $\alpha = S(\vec{p})$ starts at the tree root following the path containing it until it reaches the leaf nodes, i.e., the data pages. Search is guided by the separators in the index pages, i.e., when storing the end of regions it means that we are following the separator σ_i on the current index page with $\alpha \leq \sigma_i \wedge \nexists \sigma_k | \alpha \leq \sigma_k < \sigma_i$. The actual search for σ_i is efficiently accomplished by a binary search when index entries are stored in key order on the index pages. It will always lead to a data page which has to be fetched and its content needs to be inspected.

As the BUB-Tree contains intervals on the index pages, the algorithm has to be modified to search for an interval $[\sigma_i, \epsilon_i]$ with $\sigma_i \leq \alpha \leq \epsilon_i$. When storing the intervals on the index pages in address order, binary search can be utilized to find a matching entry efficiently. Table 6.1 shows the basic search loop.

If there is no such interval then search stops as there is no such point and no need to inspect any further pages. Otherwise, α is included in an interval and we follow the associated link to the child node. If the child node is an index page we proceed as before. If it is a data page we check whether the point exists or not by inspecting the data page content.

The worst case I/O complexity of this search is the same as for a UB-Tree, which is logarithmic to the size of the database, i.e., we need to access h pages, where h is the height of the tree. However, the UB-Tree always has to access h pages, best and worst case are the same! In contrast to this, the BUB-Tree search may abort at the root node and thus access only a single page, resulting in a best case I/O complexity of $O(1)$. Further, as higher levels of the index will usually stay in cache, the accesses are only logical page accesses, but not physical ones.

```
DataPage BUBTree::findDataPage(Tuple t)
{
  Page &page = rootPage;
  Address z = SFC(t);
  while (page.type() == PageType::Index) {
    Position i = page.findWithin(z);
    if (i.isValid()) {
      page = storage.fetchPage(page[i].link);
    }
    else {
      throw(NoDataPage);
    }
  }
}
```

Table 6.1: BUBTree::findDataPage

6.5 Insertion

UB-Tree insertion requires a point search in order to locate the region (and the corresponding data page) including the new point. The found data page is inspected and if the point is not there, it is inserted and the updated page is stored.

A BUB-Tree point search may not return a data page if the new point lies in dead space w.r. to the current index. Consequently, the search algorithm has to be modified in order to always return a data page.

There are two cases that have to be distinguished in order to decide which index entry $[\sigma_i, \epsilon_i]$ on an index page of n index entries to follow:

α is contained within an interval i: $\alpha \in [\sigma_i, \epsilon_i]$
In this case, we follow the entry i similar to the UB-Tree algorithm.

α lies in dead space w.r. to the index : $\nexists i | \alpha \in [\sigma_i, \epsilon_i]$
There are three cases when this can happen, i.e., α lies

before the first interval: $\alpha < \sigma_1$
Search proceeds with the path $i = 1$.

after the last interval: $\epsilon_n < \alpha$
Search proceeds with the path $i = n$.

in a gap between two intervals: $\epsilon_i < \alpha < \sigma_{i+1}$ with $i < n$
In order to minimize the coverage for this case, it is necessary to choose the path which is nearer w.r. to the address distance,

$\alpha - \epsilon_i < \sigma_{i+1} - \alpha$ **:** follow the link of entry i

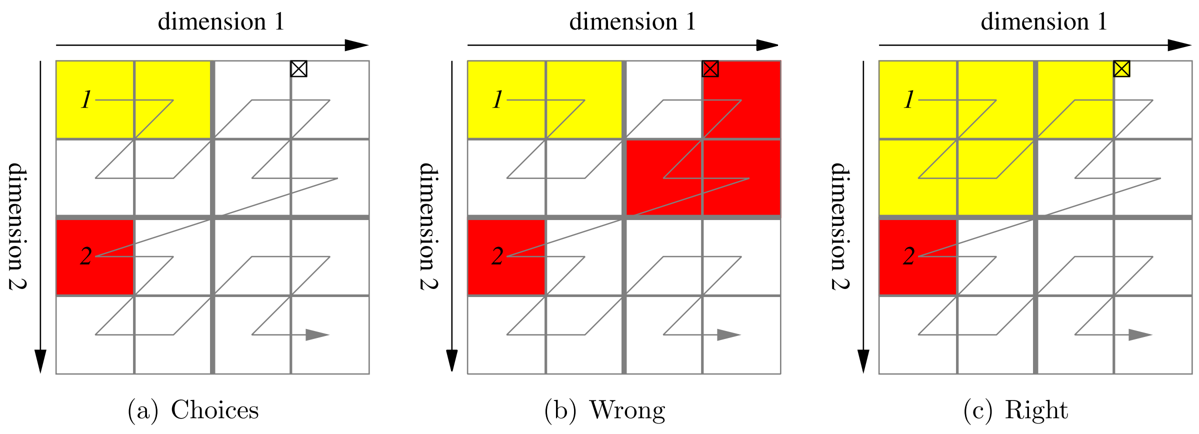


Figure 6.3: Selecting the Right Path for Insertions

$\alpha - \epsilon_i > \sigma_{i+1} - \alpha$ **:** follow the link of entry $i + 1$

$\alpha - \epsilon_i = \sigma_{i+1} - \alpha$ **:** choose the one causing fewer fringes when being extended (see Figure 6.3). This is accomplished by choosing the entry with a boundary sharing a longer common prefix with α (see also Figure 3.11 on page 50).

When reaching a data page we insert the point as it is done for the UB-Tree. A point insertion before the start σ_i or after the end ϵ_i of region i triggers an update of the index entry leading to the corresponding page. The σ_i resp. ϵ_i of the referring index entry is set to α. This might propagate up to the root page if adjustments always happen beyond the current borders of the referring index entry.

The adjustments can be efficiently performed during the traversal of the tree, i.e., when falling into a gap we adjust the index entry to include the new tuple and mark the page as dirty. The complexity is h page reads and 1 to h page writes. For random insertion the average is only slightly above a single page write. When appending to the index in address order this adjustment would occur with each insertion, but as there is no need to set $\epsilon_i = \alpha$ it is also possible to choose a greater value or the maximum thus avoiding frequent adjustments at the cost of recognizing lesser dead space.

Example 6.3 (Growth of a BUB-Trees)
Figure 6.4 shows the growth of a BUB-Tree with index and data pages having a capacity of two entries. The initial BUB-Tree has only one page, the empty root page. The first index entry and data page is created with the insertion of the first tuple. In Figure 6.4(a) there are already two tuples on the first data page and a third tuple is inserted causing a split. The resulting tree is shown in Figure 6.4(b). ◇

6.6 Page Split

In order to minimize the covered dead space in a BUB-Tree it is important to take special care of page splits caused by a page overflow during insertion or during bulk loading.

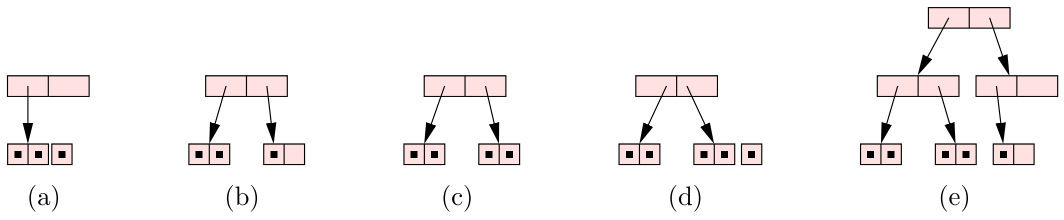


Figure 6.4: Growth of a BUB-Tree with Segments

The goal is to minimize the dead space covered by the regions of the index. Splitting in the middle of a page might not satisfy this goal, since the position of the greatest gap can lie somewhere else. However, splitting at the best position might result in a page utilization below 50% for one of the resulting pages. So, there is a tradeoff between minimizing the index-covered dead space and maximizing the page utilization. Handling this trade-off efficiently is the topic of this section.

A gap on a data page is defined as the difference of the addresses of two consecutive points $\vec{p}_i$ and $\vec{p}_{i+1}$ which are neighbors w.r. to address order, i.e., $\nexists \vec{p}_k | S(\vec{p}_i) < S(\vec{p}_k) < S(\vec{p}_{i+1})$:

$$\Delta_i^D = S(\vec{p}_{i+1}) - S(\vec{p}_i) \tag{6.1}$$

For index pages with the entries $[\sigma_i, \epsilon_i]$, a gap is defined as difference of the end of one region to the start of the next region, i.e.,:

$$\Delta_i^I = \sigma_{i+1} - \epsilon_i \tag{6.2}$$

A page with n tuples resp. index entries including the newly inserted one is processed as follows. In case of $n = 2$ the split is trivial, as there is no choice, so each tuple goes on its own page. For $n > 2$ and honoring a minimum page utilization U_{min} with $0 \leq U_{min} \leq 50\%$ it is necessary to calculate the gaps between tuples in the range $[s, e]$ where s and e are the position of the tuples on the page and are defined as follows:

$$\begin{aligned} s &= 1 + \lfloor (n-2) * U_{min} \rfloor & (6.3) \\ e &= n - \lfloor (n-2) * U_{min} \rfloor & (6.4) \end{aligned}$$

The best split is then between the tuples resp. index entries at position p and $p + 1$ with p defined as:

$$p | \Delta_p = \max\{\Delta_s, \ldots, \Delta_{e-1}\} \tag{6.5}$$

If there is more than one maximum gap, we prefer the one causing fewer fringes, similar to the path selection during insertion. Again, this can be accomplished by comparing the addresses, i.e., choosing the gap where the bounding addresses of the gap share a longer common prefix. This is basically the ϵ-split algorithm of [Mar99]. If we are not able to make a decision w.r. to fringes, we choose the gap which is nearer to the page middle, i.e., which will result in a more even page utilization.

In the worst case, i.e., for $U_{min} = 0$, calculating the best split position requires to inspect each pair of neighboring tuples. Please note, also with $U_{min} = 0$, there will always be at least one tuple on a page, since splits occur only in the gaps between tuples and U_{min} is only a threshold, but not the actual page utilization.

So far we have not addressed how to deal with the tradeoff between page utilization and covered dead space. Setting $U_{min} = 0$ can lead to a highly degenerated index w.r. to page utilization, containing only pages with a single tuple on it and one filled page.

Example 6.4 (Pathological Degeneration of a BUB-Tree)
Starting with an empty BUB-Tree we insert tuples in address order as follows. We start with $\alpha_1 = \frac{|\Omega|}{2^1}$ as the address of the first tuple. The next tuple tuple $i = 2$ has the address $\alpha_2 = \alpha_1 + \frac{|\Omega|}{2^2}$, the third $\frac{|\Omega|}{2^3}$, etc. Inserting tuples in this order and with these addresses and allowing $U_{min} = 0$ will result in splitting always between the first and second tuple and causing a split with every second insertion, after the first page has been filled to 100%. $\diamond$

The tradeoff is handled gracefully when additionally taking the following rules, into account to trigger a split at the best position instead of the page middle. $[\sigma, \epsilon]$ bounds the region corresponding to the page that should be split, consider the following measures:

1. $\frac{\epsilon-\sigma}{|\Omega|} \geq R_{min}$: Search for a best-split only if the region exceeds a certain size w.r. to the universe size. For small regions, the normal UB-Tree split is performed without causing further overhead by searching for a best split position.

2. When running a search for the best split position honoring U_{min}:

   (a) $\frac{\Delta_i}{\epsilon-\sigma} \geq G^R_{min}$: Trigger a best-split only if the gap covers more than a given percentage of the region size. If the points in a region are uniformly distributed w.r. to their addresses, then all gaps have approximately the same size $\Delta_i \approx \frac{\epsilon-\sigma}{n}$. In this case there is no need to give up the 50% page utilization guarantee and thus a 50% split is performed.

   (b) $\frac{\Delta_i}{|\Omega|} \geq G^\Omega_{min}$: Trigger a split only if the gap size exceeds a certain minimum size w.r. to the universe size. This is for fine tuning the splits, but it is not generally necessary.

   (c) If no best split is triggered, a 50% split will be performed. This preserves the page utilization guarantees of UB-Trees where possible.

The overall cost to find the best split position is bound by the number of elements n on a page and it is inverse linear to the desired page utilization, i.e., $O(1)$ for $U_{min} = 50\%$ and $O(n)$ for $U_{min} = 0$, i.e., its worst case complexity depends on the number of gaps to inspect, i.e., it is linear to $O(e - s)$.

It is also possible to limit the described split algorithm to data pages, but by doing this we would loose chances to prune search paths earlier during an index traversal.

Finally, during filling a page might have accommodated more than just one big gap. Therefore, one could allow a multi-split, i.e., not only one but $\lfloor \frac{1}{U_{min}} \rfloor$ splits. When obtaining the gaps we perform splits according to their priority, i.e., first the biggest gap, then on the two new pages another split if the region size and minimum page utilization allow it. The gaps are only calculated once, but the regions size for each new region.

6.7 Deletion

Deletion is handled by a point query and on success we delete the point from the found data page. Deletion of the first or last tuple of a page triggers an update of the index entry leading to this page when minimal dead space coverage is desired. If optimal coverage after deletion is not necessary, the bounds may also be kept as they are until a page merge occurs.

Page underflows are not triggered by a page utilization below 50%, but when it is below the minimal page utilization U_{min} of the specific BUB-Tree.

A merge should be made with the neighboring region that is nearer w.r. to address order, i.e., for a page i we compare the distance to the previous page $\sigma_i - \epsilon_{i-1}$ and the distance to the next page $\sigma_{i+1} - \epsilon_i$ and choose the page with the smaller distance. The complete algorithm for selecting the right page for a merge is given in Table 6.2.

There are two case after the merge. If the merges page has an utilization greater than 100%, a page split is performed and the referring index entries are updated.

In the other case, all tuples fit on the merged page and the index entry referring to it is corrected to the new bounds of the page and the other one is removed if all tuples fit on one page. The update and removal may propagate up to the father level of the index when they occur at the first resp. last entry of an index page. If the new page exceed the maximum page size, then it is split again and the index entries of both pages are updated.

6.8 Range Queries

Range query processing is similar to the UB-Tree as discussed in section Section 3.2 on page 40. Basically, it is a traversal of the tree in increasing address order. We need the NJI and NJO algorithm for calculating the intersections of the query shape with the SFC.

Processing of a query box $Q = [\vec{l}, \vec{u}]$ starts like a point search for the address $\alpha = S(l)$, where S calculates the SFC address. In the beginning the root page is searched for an entry containing α (Figure 6.5(a)). This is accomplished by a binary search for pages storing the entries in address order.

If no entry containing α is found, search stops at a gap, i.e., before the first index entry with a starting address σ greater than α. Consequently, this is the address for continuing the search.

```
DataPage BUBTree::mergePage(Page &page)
{
  if (page.previous == NULL && page.next == NULL ) {
    // do nothing, i.e. there are no other pages
  } else if (page.previous == NULL && page.next != NULL) {
    page.merge(page.next);
  } else if (page.previous != NULL && page.next == NULL) {
    page.merge(page.previous);
  } else if (page.start - page.previous.end <
             page.next.start - page.end) {
    page.merge(page.previous);
  } else {
    page.merge(page.next);
  }
}
```

Table 6.2: BUBTree::selectMergePage

For the further processing we need the NJI resp. NJO algorithms discussed in Section 3.2 on page 40, calculating the first intersection of the SFC with the query shape after address α resp. the first point greater or equal to α just before the SFC leaves the query shape.

There is one case that allows to proceed the traversal, i.e., if the current query interval $[nji, njo]$ intersects the interval $[\sigma, \epsilon]$ of the index entry. This is detected by testing $\sigma \leq NJO(Q, \alpha)$ (Figure 6.5(b)). If this condition is true, search continues with the node the index entry refers to.

Otherwise a new point search is started with $\alpha = NJI(Q, \sigma - 1)$. The NJI will return an address greater than the input address, but the actual nji might be σ. Thus, search has to continue just before the beginning of the region denoted by $[\sigma, \epsilon]$ (Figure 6.5(c)).

This new point search might bring us back to the current entry, but it might also go to another region skipping a bigger SFC-segment, thus checking the next query interval directly might be a waste of time.

If a data page is found, it is post-filtered for result tuples and search continues with the NJI after the end of the region corresponding to the data page.

The algorithm terminates when there is no index entry left to follow for the point search or when accessing an index entry with a start greater than the upper bound $\vec{u}$ of the query.

The complete algorithm is given in Table 6.3. It detects dead space where it is not covered by the index and thus avoids page accesses where possible.

```
void BUBTree::rangeQuery(Pipeline &pipe, const Query &Q)
{
  Page &page = rootPage;
  Address nji = S_min(Q);
  while (1)
  {
    while (page.type() == PageType::Index)
    {
      // find an entry containing nji,
      // or the first entry with (start > nji)
      Position i = page.findEntry(nji);

      if (i == NOTFOUND) return;

      Address &start = page[i].start;

      if (nji < start && NJO(Q, nji) < start)
      {
        nji = NJI(Q, start - 1);
        page = rootPage;
      }
      else
      {
        page = storage.fetchPage(page[i].link);
      }
    }

    // check page for result tuples and return them
    pipe.push(filterTuples(Q, page.content()));

    // search is finished
    if (S_max(Q) <= page.end) return;

    // continue search with NJI after the region
    nji = NJI(Q, page.end);
  }
}
```

Table 6.3: BUBTree::rangeQuery

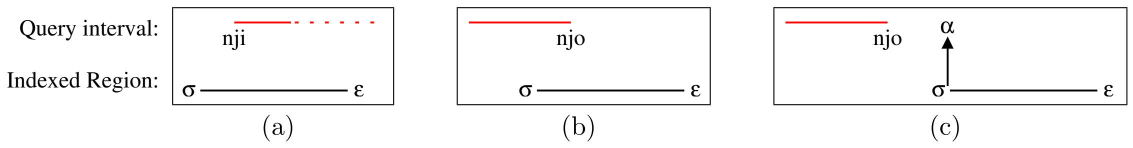


Figure 6.5: Relation of Query Interval to Index Interval in a BUB-Tree

6.9 Bulk Insertion and Bulk Deletion

In Section 4.3 and 4.4 we have addressed bulk operations on UB-Trees. Due to the different structure of the BUB-Tree they cannot be reused without modification. This section discusses the modification required to make them work for the BUB-Tree. All algorithms retain their original I/O complexity.

6.9.1 Initial Bulk Loading

Initial bulk loading is the creation of a new BUB-Tree from a data set. As for the UB-Tree, the data set is sorted according to address order and then loading starts. The UB-Tree bulk loading algorithm ensures a desired page utilization U.

The BUB-Tree goal of minimizing the covered dead space is orthogonal to this. The solution addressing both problems are the rules triggering a best split as listed in Section 6.6.

The UB-Tree writes a page if its utilization reaches U. For the BUB-Tree the page is filled completely including the tuple causing an overflow. If a best split is triggered according to the rules listed in Section 6.6, it is performed at the designated position. Search for the best split position is performed in the range of positions $[U_{min}, 100\% - U_{min}]$ of the page. If no best split is triggered, a split producing a page utilization of U is performed.

The first page w.r. to address order is written to disk and the other one is kept in main memory for further filling. Whenever a page is written its bounding interval is inserted into the index. As for the UB-Tree, only the path to the current page needs to be kept in main memory and writing of index pages is postponed until they are not subject to modification anymore, i.e., when descending into a new path.

This process continues until there is no data left.

Bookkeeping of the average page utilization allows for disabling the best split when it differs to much from the designated page utilization U.

6.9.2 Incremental Bulk Loading

Incremental bulk loading is the insertion of new data to an existing BUB-Tree. The core of the algorithm is the same as for the UB-Tree. When searching for a page to insert new data, the insertion algorithm of Section 6.5 must be used in order to ensure a data page is found while keeping index coverage low. Filling and splitting the pages is performed in the same way as described in the last section on initial loading.

No further modifications are required.

6.9.3 Bulk Deletion

Also the bulk deletion for BUB-Trees requires only minor changes to the algorithm presented for UB-Trees. First, the BUB-Tree range query has to be used instead of the UB-Tree range query algorithm. Second, page merges and tuple redistribution should take minimal coverage of dead space into account and an underflow is only triggered when the page utilization drops below U_{min}.

Further, instead of simply moving tuples to the page with the underflow until a 50% utilization is restored, again it is sufficient to only restore the U_{min} requirement.

Another solution to fix an underflow is a merge of both pages followed by a best split resp. a normal 50/50 split, if no best split position was found.

6.10 Reorganization

The index might degenerate w.r. to page utilization. An example for this has been given before. Further, the index might not be optimal when adjustments of index entries had been postponed for deletions or when bounds have been chosen to be not tight in order to avoid frequent adjustments of bounds during insertion.

Thus, it might be necessary to reorganize the index from time to time or on demand in order to enhance query performance and/or page utilization. In the following we discuss different aspects of such a reorganization separately. However, it is possible to combine them and thus avoid multiple passes over the complete index and data.

All operations presented here modify the tree only locally, i.e., they only require to lock modified pages on the currently processed path from the root to the leafs of the index and where necessary its neighboring pages. Further, it is not necessary to run them exclusively as they are not modifying data, but they can be run concurrently with other operations, e.g., queries, deletes, updates.

The operation presented here will reorganize the whole index. If the index only requires a partial reorganization, the operation can be easily restricted to the relevant range. This is done by utilizing the range query algorithm to select the pages that reorganization should be applied to.

When modifying index pages it is important to consider the trade-off between reading and writing them multiple times versus blocking other transactions. As there might be more than one modification of an index page it is a good idea to lock it until no more modifications can occur. However, locking index pages at higher index levels, or even the root, blocks transactions requiring any node in the subtree of the locked page. Therefore, as there are fewer pages on the upper levels and as they will be in cache most of the time it is advised to unlock them after modifying them and thus do not block too many other transactions.

6.10.1 Tightening Bounds

Optimal bounds can be restored by adjusting them during a depth-first index traversal. Index entries are adjusted to the minimum and maximum of their child nodes in a bottom up process, i.e., when reaching a data page the index entries on the path to it are corrected and modified index pages marked as dirty[1]. When following a new path all dirty pages on the old path are written to disk as they are not subject to modifications anymore. The cache needs to hold h pages where h is the height of the index.

All index and data pages are read once and modified pages are written only once. As the depth-first search is done in address order pre-fetching is also utilized when working on a bulk loaded BUB-Tree.

6.10.2 Optimizing the Index Coverage

Minimizing the index coverage is a goal which is not accomplished easily, thus for us optimizing refers to a reduction of the index coverage.

Looking at two regions $[\sigma_i, \epsilon_i]$ and $[\sigma_{i+1}, \epsilon_{i+1}]$ which are neighbors w.r. to address order, it is simple to decide if another partitioning would be better. If a search for a best split positions in the range $[\sigma_i, \epsilon_{i+1}]$ finds a position which is different to the current split position, then we have found a better partitioning. Of course, U_{min} should be taken into account to limit the search on those gaps that will ensure U_{min} in case of a split. U_{min} can be changed anytime and thus may differ from the initial U_{min} of the BUB-Tree.

Applying this to the data pages means they are scanned in sequential order, always considering a pair of consecutive pages and their corresponding regions. Whenever a better split position is found, it is used and the left page, which is the one corresponding to the region with the smaller address, is stored and the index entry referring to it is updated. The right page is not written immediately as it needs to be considered in the next cycle of the loop which might modify it again. Index pages are marked as dirty and are kept in cache until displaced from it. Again it is only necessary to have a small cache, i.e., a cache of $h + 1$ pages is sufficient.

If data pages are linked to each other, this algorithm will read all data pages in sequential order, but only those index pages lying on a path to a modified data page. The path to a data page is found simply be performing a point search on one of the points stored on the data page. Modified pages are written only once.

The resulting BUB-Tree has the same number of pages, but if any modifications have been made, reduced coverage allowing for pruning more search paths.

[1]A page marked as dirty will be written to disk before displacing it from the cache. This is a common strategy to postpone writing to disk until it cannot be avoided anymore.

6.10.3 Increasing the Page Utilization

Performing a best split causes in the worst case the creation of pages all having a page utilization of U_{min}. Thus, increasing the average page utilization and reducing the number of pages usually has a positive effect on query performance as there are fewer pages which can be accessed.

However, simply merging pages or redistributing tuples might cause the degeneration of the BUB-Tree performance, as possibilities to prune search paths might be reduced. Thus, only looking at the page utilization of a single page is no solution, but the average page utilization should be increased.

Therefore, there is only one way to increase the page utilization, i.e., by merging two consecutive pages which both have a page utilization below 50%. Further this should only be done if there is no best split position between the tuples on those two pages, i.e., in range $[\sigma_i, \epsilon_{i+1}]$. When a merge occurs, the index entry referring to the deleted page is removed and the other one is adjusted to its new bounds.

This algorithm will read all data pages once in sequential address order and all index pages lying on a path to modified data pages. It will only write modified pages and all of them just once.

6.11 Implementation-Variants

While the concept of the BUB-Tree is a set of intervals, it is not necessary to actually store the intervals. An alternative is given in the following. As for the UB-Tree, only the end of a region is stored, i.e., only the separators. Additionally, a special link is introduced, i.e., instead of referring to a child node it is invalid signaling that it refers to dead space.

Applying this to the BUB-Tree requires to decide when to use this special link type. Using it always will result in a tree equivalent to a normal BUB-Tree storing SFC-segments. Thus, the invalid links should only be used if there is a reasonable amount of dead space.

Detecting this is already solved, i.e., when triggering a best split there was a reasonable segment of dead space. In this case, the best split is performed, two new index entries are added. For the resulting regions $[\sigma_i, \epsilon_i]$ and $[\sigma_{i+1}, \epsilon_{i+1}]$, the following index entries are generated:

$$\begin{aligned} \text{new link to new page} &: [\epsilon_i, pagenumber(newPage)] && (6.6)\\ \text{optional dead space link} &: [\sigma_{i+1}-1, invalid] && (6.7)\\ \text{unmodified link to old page} &: [\epsilon_{i+1}, pagenumber(oldPage)] && (6.8) \end{aligned}$$

This increases the coverage in comparison to a BUB-Tree storing segments, whenever no best split was performed, but on the other hand more index entries are possible in this case, where probably no significant performance advantage would be gained by pruning search paths. With more index entries the fanout of the tree is higher and thus its height might be reduced.

Also without compression, addresses require generally more storage than the page link and thus the size of three index entries encoding two regions and the dead space between them (three addresses and three links) is smaller than storing them as two SFC-segments, i.e., four addresses and two links.

There are two choices for storing the dead space pointers.

1. They are only stored on the last level of index pages just before the data pages. While this is easier to maintain it also allows to prune search paths only at this level.

2. Dead space pointers are pushed to upper levels of the index. When splitting a page, dead space at the start or the end of page is pushed as an index entry to an invalid page in the father page.

For uniformly distributed data which was bulk-loaded, the resulting BUB-Tree would be identical in structure and size to a UB-Tree on the same data.

The adaption of point search, insertion, and range query is strait-forward. The search for an address α is a standard B$^+$-Tree/UB-Tree point search with the following exceptions, when hitting an index entry $[\sigma_i, INVALID]$ with $\alpha \leq \sigma_i \wedge\ (\nexists \sigma_j | \alpha \leq \sigma_j < \sigma_i)$ we do the following in case of a:

Point Search: search aborts with "point not found".

Insertion: The distance $\Delta = \alpha - \sigma_{i-1}$ to the previous link is calculated and if smaller than $\sigma_i - \alpha$ search follows the entry $i-1$ otherwise it follows the entry $i+1$, i.e., the next index entry. If there is no previous or next index entry on the same level, then the other existing one is chosen for continuing the tree-traversal.

Range Query: If the $NJO(\alpha) > \sigma_i$ search follows the entry $i+1$, otherwise a new point search is started with $\alpha = NJI(\sigma_i)$.

Implementing the BUB-Tree in this way, eases the index selection problem for the user, i.e., one can start with a BUB-Tree being identical to a UB-Tree in structure, size and performance.

When encountering performance problems due to queries on dead space, the index can be tuned without rebuilding it or going offline. Tuning the index is: Decreasing the minimum page utilization U_{min} and the thresholds for triggering a best split followed by an online reorganization.

One may even limit the reorganization to a specific range of the universe covered by the queries causing the major performance problems. This limits the reorganization cost and addresses specific query requirements.

Example 6.5 (Growth of BUB-Trees with separators)

Figure 6.6 shows a BUB-Tree with separators instead of intervals having the same structure as Figure 6.4. A $\boxtimes$ in Figure 6.7(b) represent the invalid links representing dead space. It is not necessary to store a separator for dead space as last index entry as it is implicitly defined by the last separator σ to an indexed region, i.e., $[\sigma + 1, S_{max}]$ must be dead space. $\diamond$

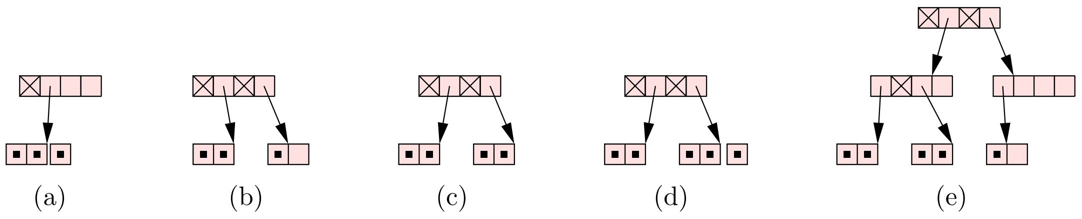


Figure 6.6: Growth of a BUB-Tree with Separators

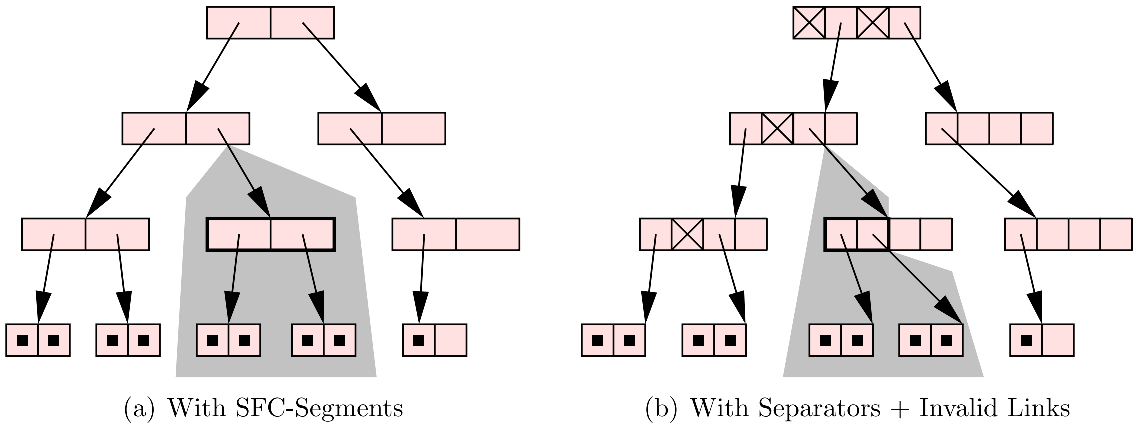


Figure 6.7: Structure of BUB-Tree Variants

Example 6.6 (Comparison: BUB-Trees with Segments vs. Separators)
Figure 6.7 depicts the different tree structures for a simple example with 9 data points and a page capacity of 2 points for data pages. Index pages contain two SFC-segments + two links (Figure 6.7(a)) resp. four SFC-addresses + four links (Figure 6.7(b)). The data is the same as for the example in Figure 6.1 on page 126. The little black squares in the leafs represent the tuples on the data pages. The dead space between the data page three and four is small and thus no best split was triggered resulting in only two separators, but no separator with an invalid link (see the shaded areas in Figure 6.7). ◇

6.12 Comparison of BUB-Tree and R-Tree

In the following we compare the BUB-Tree to the R-Tree [Gut84] and its variants [SRF87, BKS+90], which are accepted as multidimensional indexes with good query performance and have been already integrated into several DBMSs.

Most papers are only considering the query performance, neglecting other essential aspects, i.e., the maintenance cost, the space requirements, concurrency, etc. As these are essential for actual applications and a DBMS integration, they should also be addressed. The success of the B-Tree is based on addressing all of these aspects very efficiently. {R}-

Trees are often denoted as the multidimensional extension of the B-Tree, while this is true w.r. to being balanced multi-way trees, we will point out that the {R}-Tree cannot be claimed as multidimensional extension of the B-Tree w.r. to performance.

The R-Tree and its variants R^*-Tree and R^+-Tree are based on a hierarchy of minimal bounding boxes (MBBs), often also referred to as minimum bounding rectangles. Like the B-Tree, it is a balanced tree and maps tree nodes to disk pages. Its design flaw is a possible overlap of MBBs and thus it cannot provide the logarithmic performance guarantees of B-Trees for the basic operations of insertion, deletion, and point search. The R^+-Tree [SRF87] avoids the problem of overlap by clipping and redundantly storing of objects. The R^*-Tree approach to minimize overlap of MBBs, is an enhanced split algorithm combined with forced reinsertion, which achieves a partial reorganization of the tree. While this reduces the overlap it increases the average cost of insertions, but still overlap cannot be avoided.

Heuristics minimizing the overlap of bulk loaded R-Trees are the packed Hilbert-R-Tree [KF93], STG [LEL97], and the TGS R-Tree [GLL98]. Recently a new version of R-Tree, the *Priority R-Tree* (PR-Tree) was introduced by [ABH$^+$04] claiming to be practically efficient and worst-case optimal. Basically, it guarantees an overlap-free partitioning of the data pages by utilizing a k-d-Tree resp. similar partitioning techniques. In order to guarantee overlap-freeness for spatial objects, the object-MBBs in the d dimensional *work* space are mapped to a point in $2 \cdot d$ space, called the *configuration* space.

However, while the PR-Tree can provide a better partitioning than the other bulk loading heuristics, the essential problem of R-Trees remains: When inserting into an existing tree, the theoretical performance cannot be guaranteed anymore and its practical performance might suffer as well. The authors claim that updates might be done efficiently by a dynamic version of the PR-Tree using a logarithmic method, but this remains open as future work.

During our comparison we will use the following naming conventions. If a statement applies to all variants of R-Trees (R-Tree, R^*-Tree, R^+-Tree, PR-Tree), we will use *{R}-Tree.* If our arguments only apply to a specific variant of R-Tree, we will explicitly name the variant.

6.12.1 Related Work

[Ram02] presents the results of a detailed theoretical and experimental comparison of R^*-Trees and UB-Trees. For the sake of completeness, we provide his measures and summary here.

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

Regarding query performance, the UB-Tree outperforms the R^*-Tree for point and range queries. Only so-called dead-space queries, i.e., queries into the empty part of the universe, are usually processed faster by the R^*-Tree than by the UB-Tree.

6.12.2 {R}-Tree coverage

For comparisons with the BUB-Tree we also define the coverage of a R-Tree:

Definition 6.4 (R-Tree-Coverage)
The *coverage* of a R-Tree with the bounding boxes $\mathbb{R}(\Omega) = \{[\vec{l_0}, \vec{u}_0], \ldots [\vec{l_k}, \vec{u}_k]\}$ with $\vec{l_i}, \vec{u}_i \in \Omega$ is defined as $cov(\mathbb{R}(\Omega)) = \left|\{\vec{p} \in \Omega | \exists [\vec{l}, \vec{u}] \in \mathbb{R}(\Omega) : \vec{p} \in [\vec{l}, \vec{u}]\}\right|$ $\diamond$

This does not include the overlap so, we additionally define the sum of the coverage of each MBB on the lowest index level of the R-Tree as:

Definition 6.5 (R-Tree-Sum-Coverage)
The *sum-coverage* of a R-Tree with the bounding boxes $\mathbb{R}(\Omega) = \{[\vec{l_0}, \vec{u}_0], \ldots [\vec{l_k}, \vec{u}_k]\}$ is defined as $sumcov(\mathbb{R}(\Omega)) = \sum_{i=1}^{k} \left|[\vec{l_i}, \vec{u}_i]\right|$ where: $\left|[\vec{l_i}, \vec{u}_i]\right|$ is defined as $\left|\{\vec{p} | \vec{p} \in [\vec{l_i}, \vec{u}_i]\}\right|$, i.e., the number of points in Ω in the MBB resp. the volume of the MBB. $\diamond$

Considering the coverage on higher levels of the index is not so important as it cannot avoid data page accesses, but only index page accesses. Further, the upper levels of a tree based index will remain in cache thus index page accesses usually cause fewer or no I/Os.

In the following we will discuss the differences of both indexes in detail, always focusing on one aspect and finally giving a summary. We will also discuss the measures taken into account by [Ram02]. We use n to denote the number of element on a page, d for the number of dimensions, U for the desired page utilization, and h for the tree height.

6.12.3 Bounding Technique

Both indexes use bounding to reduce the coverage of the index. The BUB-Tree uses disjunct SFC-segments while the R-Tree uses MBBs which may overlap. The following holds for the coverage of the index:

$$\mathrm{BUB-Tree} \; : \; cov(\mathbb{B}) \leq |\Omega| \quad (6.9)$$
$$\mathrm{R-Tree} \; : \; cov(\mathbb{R}) \leq |\Omega| \quad (6.10)$$

But as MBBs may overlap the $sumcov(\mathbb{R})$ can be higher and may exceed $|\Omega|$. While $cov()$ is a good hint for the performance of dead space queries, $sumcov()$ hints on the performance of queries on index covered space, i.e., for $sumcov(\mathbb{R}) \leq |\Omega|$ the performance should be reasonable due to fewer overlaps, but for $sumcov(\mathbb{R}) \gg |\Omega|$ one can expect a serve performance degeneration, for both point and range queries.

Choosing MBBs offers more freedom than choosing SFC-segments. However, this freedom does not come for free during page splits, as discussed in the next section.

However, MBBs have an inherent advantage: They are axis-parallel and thus align nicely with range queries which are also axis-parallel. In contrasts to this, SFC-segments have fringes and may be bloated and this may cause additional intersections with query boxes, which do not occur with MBBs.

6.12.4 Supported Data Types

The UB-Tree and BUB-Tree are multidimensional index structures for point data. As the {R}-Tree uses MBBs is can also store extended objects. However, as discussed in the next chapter, also the UB-Tree and BUB-Tree are able to handle spatial data efficiently when transforming it and the queries into points.

6.12.5 Dimensionality

Both indexes are not restricted to a certain dimensionality, but the curse of dimensionality affects their performance.

The UB-Tree/BUB-Tree will provide its logarithmic performance for the basic operations without dependency on the number of dimensions, but range queries will degenerate with increasing dimensionality, see [Mar99].

The {R}-Tree was designed for spatial applications, i.e., two- and three-dimensional data. Both, its point and range query performance will decline with increasing dimensionality, due to increasing overlap. The higher the dimensionality, the harder it becomes to avoid overlap.

6.12.6 Multi-User Support

As already stated by [Ram02], the UB-Tree/BUB-Tree benefit from building upon the B-Tree. They can utilize B-Trees, locking and logging and, as basic operations are local w.r. to the tree, concurrency is handled gracefully.

While various concurrency control algorithms have been proposed for R-Trees [NK94, KB95, CM99] or for the more general GIST framework [KMH97], the R*-Tree concept of reinsertion severely reduces the degree of concurrency that can be achieved.

Thus, there is a tradeoff between minimizing overlap in order to provide good performance and minimizing locked pages to allow better concurrency.

6.12.7 Page Split

The R-Tree variants basically differ in the way how MBBs are chosen to minimize overlap. [Gut84] originally proposed three different algorithms with different CPU complexities, i.e., a linear $O(dn)$, a quadratic $O(dn^2)$, and an exponential algorithm $O(2^{n-1})$, where the quadratic algorithm has shown a good trade-off between run time and minimizing the overlap. Still the quadratic split shows a degeneration of performance and thus the R*-Tree added a split algorithm with a complexity of $O(2dn \log n + 6 \cdot (n - 2U - 2))$. [BPT02] propose to use a k-means clustering algorithm and a multi-way split, i.e., a spilt into k pages is allowed in order to minimize the overlap. This causes a page utilization of $\approx \frac{1}{k}$, thus no 50% guarantee can be given anymore. While the clustering is easy for point data, special distance functions are required for spatial data. Further, the authors leave it open, how to choose the seeds for the clusters.

In contrast to this, the page split of the BUB-Tree is strait-forward. Its complexity is bound by $O(n)$ and even reduced by a desired page utilization $U > 0$, i.e., for 50% it is $O(1)$, we simply split in the middle. There is no dependency w.r. to the dimensionality d.

6.12.8 Index Size

The index size, i.e., the number of index pages, depends on two factors:

- Size of an index entry $[link, \ldots]$
- The page utilization

Size of an index entry is bound by the d, i.e., the number of indexed dimensions. The {R}-Tree uses a MBB $[\vec{l}, \vec{u}]$, while the BUB-Tree uses a segment on the SFC $[\sigma, \epsilon]$. In the worst case both require the same size, since the address $S(\vec{p})$of a point $\vec{p}$ is only another representation of it.

Taking an actual implementation of storing MBBs into account, a point is stored as a vector of integers or floats. For dimensions not requiring the complete domain also smaller types may be used decreasing the requirements. In order to efficiently process them, values will be byte to be directly used without modification. Often, they even will be word aligned in order to allow for directly comparing their binary representation. Thus, between each attribute and index entry some storage will be wasted.

On the other hand, an address is a bit-string and its length corresponds to the sum of bits required to represent each dimension. Thus for lower cardinality dimensions fewer bits will be used, resulting in a shorter address. Address comparisons do not require word alignment as they are just string comparisons and thus even more space can be saved.

Thus the following holds:

$$sizeof([\sigma, \epsilon]) \quad \leq \quad sizeof([\vec{l}, \vec{u}]) \tag{6.11}$$

As cardinalities of dimensions are usually not equal and often differences are big, thus the BUB-Tree will usually result in a smaller index with a higher fanout and it even may have a smaller tree height.

As the minimum page utilization U can be controlled for both indexes, it does influence the index size. With $U = 50\%$, the index size will be similar as for the B-Tree, but dead space coverage will not be optimal. Decreasing $U < 50\%$ will increase the index size but also decrease the coverage of the indexes and the overlap of the R*-Tree, thus query performance will increase!

6.12.9 Maintenance Performance

Maintenance performance depends on:

- Point search

- Locality of page split and page merge resp. page underflow handling

When providing complexities we usually include constants where possible in order to bound them more exactly.

Point search: Insertion and deletion for the BUB-Tree are both bound by the I/O complexity $O(h)$, and it might be $O(1)$ for deletion in the best case when all page reads are served from cache causing only a write. As index entries are stored in address order binary search can be utilized, thus requiring $O(2log_2n)$ comparisons in the worst case. If finding a match or gap earlier the cost will be even lower. The factor of 2, stems from comparing the address with each bound of the SFC-segment, while for the UB-Tree there is only one comparison with the separator [Ram02].

As R-Tree index entries are not sorted, all entries of an index page have to be inspected during search, thus its cost is always $O(n)$.

Considering the I/O performance without a split or merge, the BUB-Tree requires h page reads and one data write, and when adjusting the bounds of index entries, in the worst case $h-1$ index page writes. Thus the worst case I/O performance is $O(2h)$.

Without overlapping MBBs, the R-Tree also requires $h+1$ I/Os. However, with overlap in the worst case the whole index has to be read. Only looking at each intersection MBB ensures there are no duplicates during insertion and for deletion all intersecting nodes have to be visited anyway.

Page Split I/O: The actual page split has been discussed before, so we are focusing on the I/O here. Without reinsertion, the caused I/O is the same for both indexes.

After splitting a page the two resulting pages are written and one index entry is updated and a new one inserted into the parent node. If there are no further splits and adjustments, this causes 3 page writes. If adjustments propagate up to the root, $h+1$ writes are necessary. If page splits propagate to the root, it results in the worst case, i.e., $2h+1$ page writes, two on each level and one for the additional root page.

However, the reinsertion of R*-Trees causes additional I/O, i.e., a random insertion for each reinserted tuple, where the number of reinserted tuples is determined by the reinsertion rate e.g., $n \cdot 30\%$.

Page Underflow I/O: To the best of our knowledge, the only paper addressing a page merge for the R-Tree is the original paper [Gut84]. The proposed solution for handling page underflows is the removal of the page with the underflow and reinsertion of its tuples. This causes the I/O and CPU load for $\lfloor n \cdot U \rfloor - 1$ tuples, in worst case reading the whole index for each or causing a page split for each.

In contrast to this, page underflows for the UB-Tree and also the BUB-Tree are handled gracefully, i.e., they are local operations. An underflow will be handled by moving tuples from to a neighboring page to the page with the underflow. If all tuples fit on one page, the other page can be deleted. In best case three page writes are necessary, two for the data pages and one for the index page requiring adjustments of the bounds to the data pages. In worst case $2h-1$ pages have to be written, i.e., the data pages belong to different subtrees only sharing the root page.

Summary: Taking all this into account, the {R}-Tree cannot be regarded as the multidimensional extension of the B-Tree! It fails w.r. to maintenance! It is not suited for dynamic applications, as already pointed out by [Ram02]!

6.12.10 Query Performance

[ABH+04] introduces with the PR-Tree the first R-Tree variant with guaranteed worst case performance. For a result set size of r and k data pages it causes in worst case $O(k^{1-\frac{1}{d}} + \frac{r}{n})$ data page accesses. A d dimensional PR-Tree tree maps MBBs to $2d$ space and utilizes a kd-Tree for an overlap-free partitioning. Levels of the kd-tree [Ben75, Ben79] are split along successive dimensions at the indexed points. Thus the upper bound comes directly from this partitioning and the PR-Tree is the mapping of the kd-Tree to secondary storage. However, the PR-Tree is not meant for dynamic applications, i.e., performance guarantees cannot be given anymore after random insertions.

Standard R-Trees cannot provide better worst case guarantees, but that all pages might have to be read. This holds for both, point and range queries and stems from the possibility of overlapping MBBs. In best case only one page has to be read, the root page. The average performance highly depends on the actual overlap of MBBs and the partitioning of the data.

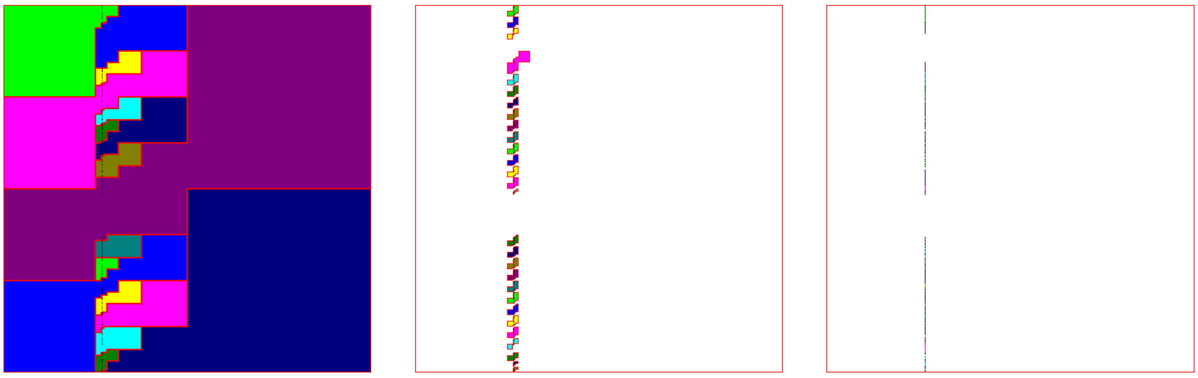
For the BUB-Tree, a point query requires at most h page reads, i.e., one path to a data page and in the best case a single page read, i.e., the point does not exist and only the root page is accessed. For range queries in the worst case the complete data base has to be read once, even for an empty result. This happens when a query decomposes into a set of SFC-segments intersecting all regions of the tree.

Example 6.7 (Pathological UB-Tree/BUB-Tree)

A UB-Tree/BUB-Tree requires to access all pages, if all data points are located along a hyper plane and the query is also a hyper plane of the same orientation, but slightly shifted. However, a query corresponding to an orthogonal hyper plane will only intersect a single region thus causing h page accesses.

This data distribution is pathological as points do not differ w.r. to one or more dimensions. Therefore, one should reduce the dimensionality of the UB-Tree/BUB-Tree, i.e., attributes which do not differ should not be indexed. If there are more hyper planes containing points, the partitioning will enhance, as regions will extent and will not be aligned w.r. to the hyper plane of the data anymore.

Also, selection dimensions for indexing usually requires to take the query workload into account, i.e., only restricted dimensions should be indexed.



(a) UB-Tree (b) BUB-Tree (c) R*-Tree

Figure 6.8: Partitioning of Skewed Data for different Indexes

Figure 6.8 depicts such a data distribution for UB-Tree, BUB-Tree and R*-Tree. The R*-Tree in this figure consists of a stack of non overlapping MBB. In fact the MBBs of the R-Tree are just line segments. ◇

6.13 Experiments

In the following we present experiments covering the different aspects of performance. The tested indexes are the UB-Tree, BUB-Tree and R*-Tree as implemented in RFDBMS (Section 4.2.3 on page 68). All indexes use the same code for tuple handling, pages processing, post filtering of data page content, parsing of flat files, etc., and differ only in their actual tree structure and related algorithms. An overview of the labels used for measures is in Appendix C.

Where not explicitly stated, BUB-Tree and R*-Tree have a minimum page utilization of 40% conforming to [BKS+90] in order to allow them to perform better partitioning of the data during a split. The BUB-Tree thresholds (Section 6.6 on page 132) were set in percentage of the address, e.g., $R_{min} = 50\%$ where 50% corresponds to the prefix of the address with size 50% w.r. to the address length. Specifying the threshold for the region size w.r. to the address prefix is more convenient than in relation to the universe and there is a one to one relation between them. Each bit in the address corresponds to halving the data space w.r. to its parent space. A bets split is only considered for regions that have at least one bit set in their prefix. The other BUB-Tree thresholds were: $G^R_{min} = 50\%$ and G^Ω_{min} was ignored.

For the experiments we used a BUB-Tree implementation within RFDBMS. Z-regions were stored as intervals in the index pages, i.e., index entries look like $([\sigma, \epsilon], pagelink)$. Best-splits were only used for data pages, but not for index pages. Chances to prune much dead space become fewer in higher levels of the index, since only gaps between lower level nodes can occur. Thus allowing a best-split for index pages would not provide any significant increased dead space pruning abilities, but only increase the number of index

Label	Description
ub =	UB-Tree
bub =	BUB-Tree
rs =	R*-Tree without reinsertion
rs-30 =	R*-Tree with reinsertion of 30% conforming to [BKS+90]

Table 6.4: Labels used for indexes

pages and thus increase the cache requirements. Therefore, we have decided to avoid this and keep the number of index pages small.

Indexes are labeled as listed in Table 6.4.

6.13.1 Random insertion

Random insertion is a typical maintenance operation for applications where the data changes dynamically. As discussed before the {R}-Tree suffers here from allowing overlap.

6.13.1.1 Uniform Data Distribution

The first set of measurements is for a data set of 5000 tuples with an uniform data distribution and dimensions from 2 to 9 with a domain of $[0 : 16383] = [0 : 2^{14} - 1]$. It has been performed on the system described in Appendix A.3.

Each tuple additionally had an identifier, i.e., a string of 6 bytes. Page size was set to 512 byte and a cache of 100 pages was used. Due to this small setup of page size and database, all measured times are CPU bound, but not I/O bound. Nevertheless, we also provide physical page accesses as a measure, as they are indicating the cache locality of the indexes.

With increasing dimensionality the key length increases and thus the cost for key calculations and comparisons is increased, while fewer entries fit onto one page.

Therefore, all indexes show a dependency w.r. to the dimensionality. Figure 6.9(a) depicts the elapsed times. *ub* and *bub* show a slight increase with dimensionality caused by the longer keys. The {R}-Trees show a steep increase, which is cause by additional logical reads as depicted in Figure 6.10(c) on page 152. The additional logical accesses stem from overlapping MBBs. *rs-30* with reinsertion causes even more page accesses and thus requires longest.

Page Utilization: All indexes maintain an average page utilization of $\approx 70\%$ (Figure 6.9(b)). Due to the different partitioning and page capacities, splits occur at different times. As tuples are randomly inserted, pages fill-up and at sometime most pages are filled and page splits will become more frequent causing a drop down in the average page utilization. When insertions continue, the page utilization will increase again, thus we omit the average index page utilization here. *rs-30* with reinsertion, does best here, by redistributing tuples before causing a split.

Page Capacity and Index Page Compression: The actual index page capacities are depicted in Figure 6.9(d). In order to fit more tuples onto a page it is possible to compress its content. This will increase CPU cost, but reduces I/O due to fewer index pages and a higher fanout.

Figure 6.9(c) shows the average compression ratio for the index pages of the different indexes. We have simply compressed the binary page images by the LZW algorithm [Wel84].

ub shows a nice property of its split algorithm. In order to avoid fringes it chooses an address which corresponds to the highest possible level of the "quad-tree" partitioning (see also Figure 3.11 on page 50). As a side effect, the resulting addresses have mostly trailing 1-bits. With increasing dimensionality addresses become longer, but at the same time the universe increases exponentially and thus the trailing 1-bits usually become more w.r. to the addresses length. Thus, for *ub* the size requirements of compressed index pages decreases with dimensionality. For two dimensions the compressed index page has a size which is only $\approx 50\%$ of the original page, then the compressed size decreases and reaches $\approx 35\%$ at dimension seven. Also {R}-Trees allow a reasonable compression. This is caused by the limited domain of the dimensions using only the lower half of the binary integer word and leading to approximately 50% 0-bytes on the binary image of the index page. *rs-30* with reinsertion has a higher page utilization causing fewer 0-bytes and thus does not achieve the same compression as the R*-Tree without reinsertion. *bub* is not able to provide the same compression, as its addresses are bounding addresses and thus do not have trailing 1-bits, but they only share a usually much shorter common prefix.

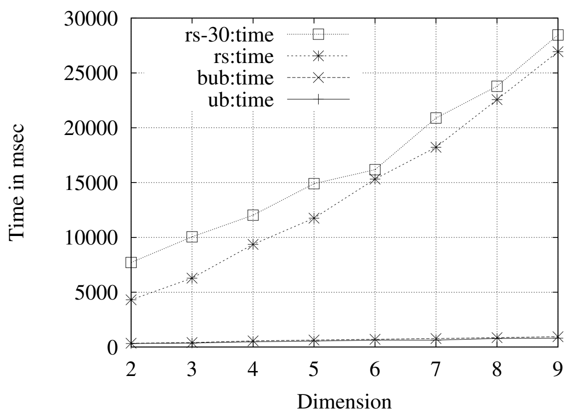
However, with a greater domain for the dimensions the compression for *rs* drops, while *ub* and also the BUB-Tree benefit from it due to longer address suffixes resp. prefixes.

Page Numbers and Index height: Figure 6.9(e) shows the number of created index pages and Figure 6.9(f) the number of data pages. The number of index pages increases for all indexes in correlation to their index page capacity (Figure 6.9(d)). *rs-30* with reinsertion is able to keep the numbers lower due to delaying splits by redistributing the tuples. *bub* has about twice the index pages, which correlates to its doubled key length.

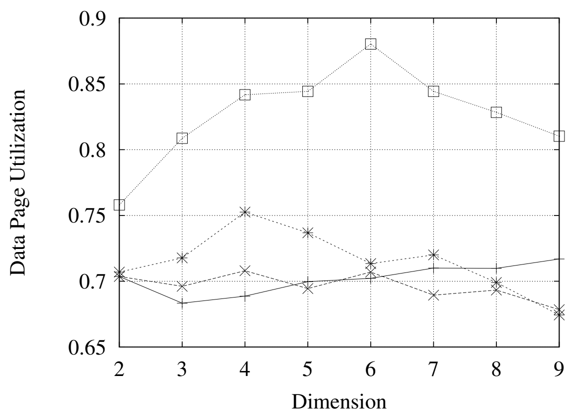
The number of data pages (Figure 6.9(f)) is very similar for all indexes, except for *rs* with reinsertion. Again this is caused by the delayed splits which come along with a increased average data page utilization as depicted in Figure 6.9(b).

Finally, the index heights are depicted in Figure 6.10(b)

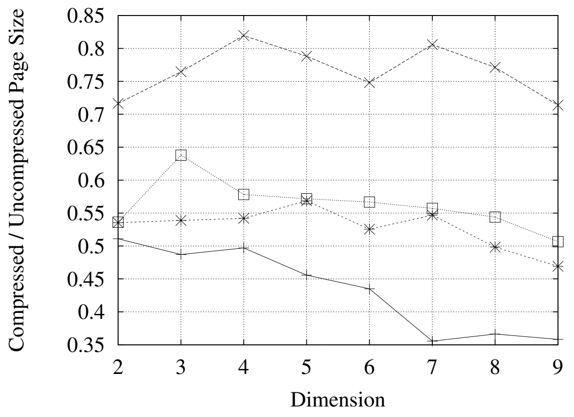
Index Coverage: The reinsertion of *rs* reduces the number of required pages by redistributing tuples to other pages before actually causing a split. This increases the average page utilization and keeps the number of pages lower. However, it also avoids further partitioning of the data and thus the overlap of MBBs increases. Reinsertion has just the opposite effect here, instead of reducing the overlap it increases it even further. By observing this phenomenon during the growth of the index, we discovered that choosing the right subtree during an insertion is suboptimal. The algorithm chooses the path only w.r. to



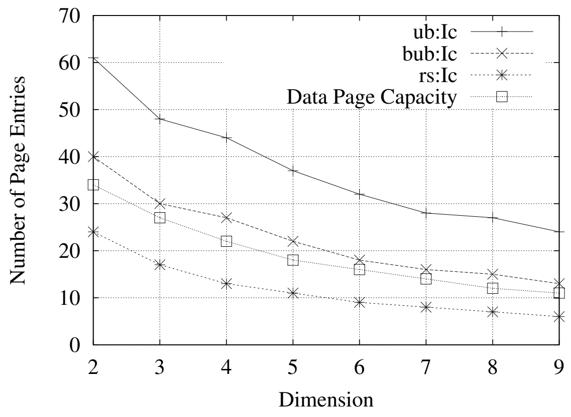
(a) Time



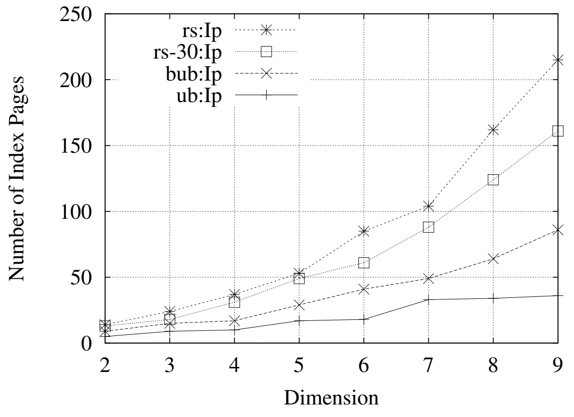
(b) Data Page Utilization



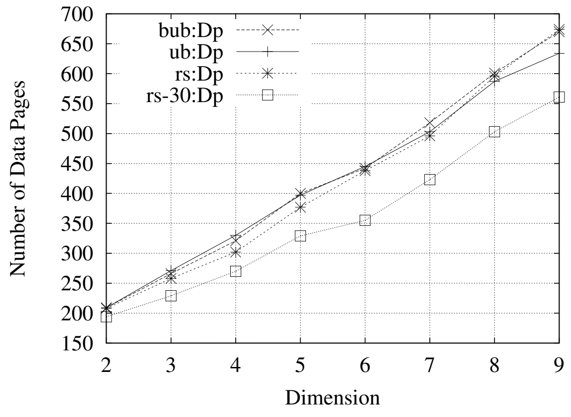
(c) Index Page Compression



(d) Index/Data Page Capacity

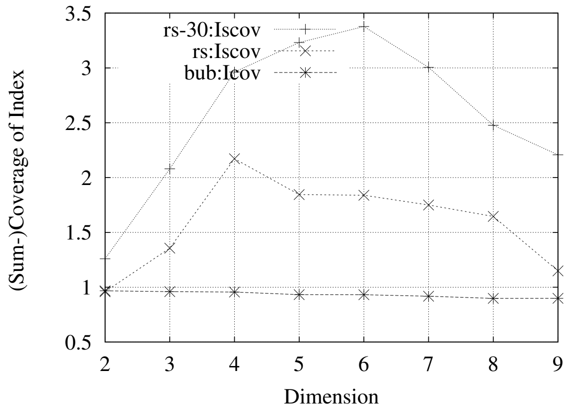


(e) Number of Index Pages

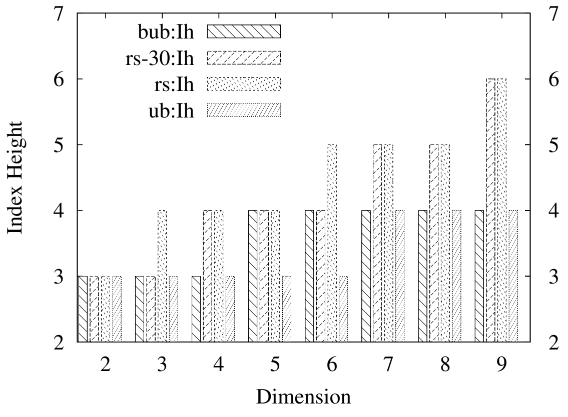


(f) Number of Data Pages

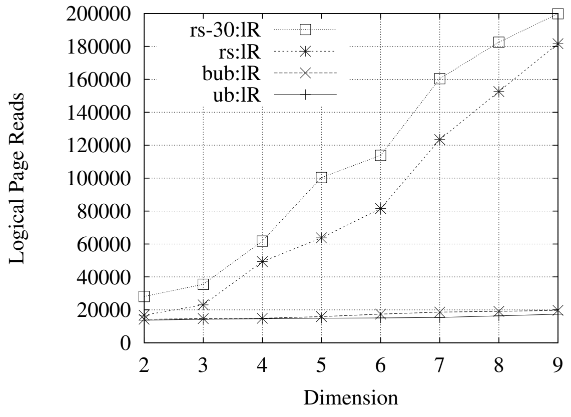
Figure 6.9: (Uniform) Random Insertion: Time and Size
Line and point types are the same in all plots.



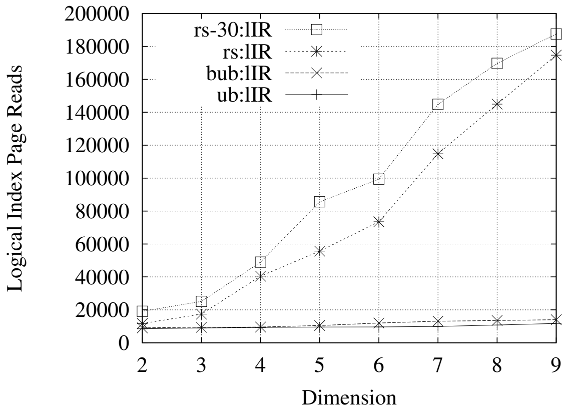
(a) Index (Sum-)Coverage



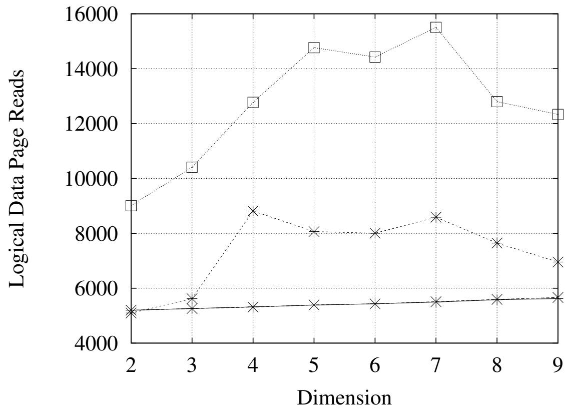
(b) Index Height



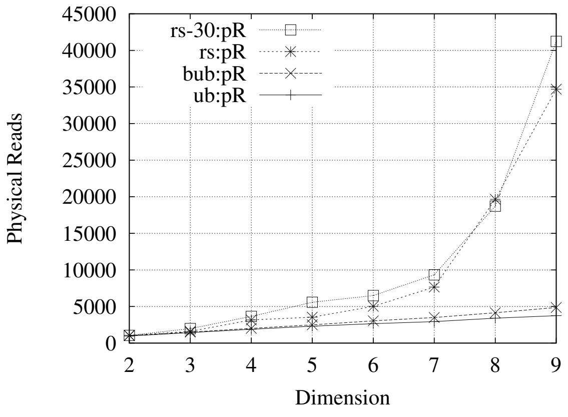
(c) Logical Reads



(d) Logical Index Page Reads

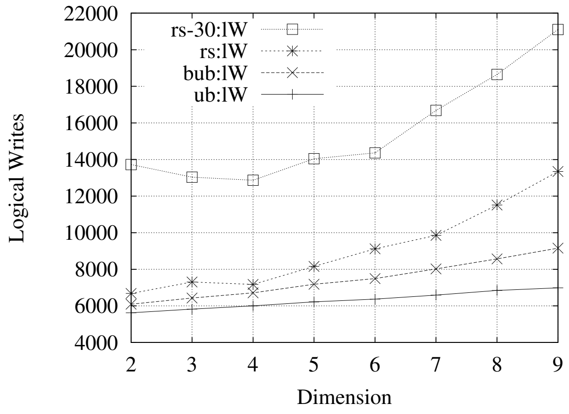


(e) Logical Data Page Reads

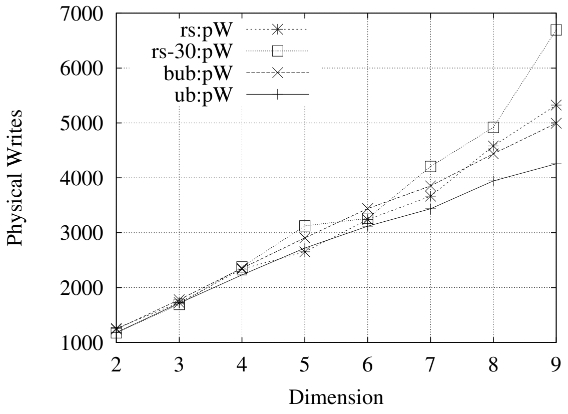


(f) Physical Reads

Figure 6.10: (Uniform) Random Insertion: Index Coverage and Height, and Page Reads Line and point types are the same in all plots.



(a) Logical Writes



(b) Physical Writes

Figure 6.11: (Uniform) Random Insertion: Page Writes

the current node, selecting the one causing the least MBB=overlap and requiring the least MBB-enlargement. This is done during traversal and may lead to additional overlap. The only way to circumvent this would be an exhaustive search, going down all relevant paths and deciding at the last index level which data node is the best. But this would further increase the cost of insertions.

Figure 6.10(a) on the preceding page shows the *sumcov*($\mathbb{R}$) of the {R}-Trees, which increases with the dimensionality and then drops again. The actual coverage for the {R}-Trees is similar to the coverage of the BUB-Tree, which stays stable just below 100% as guaranteed by the BUB-Tree design. A coverage of nearly 100% is not remarkable here, since the data is distributed uniformly. Remarkable is, that the {R}-Trees have a coverage significantly higher than 100% for higher dimensions.

Page Reads: Figure 6.10(c) on the facing page shows the logical reads, mainly caused by index reads as can be conducted from Figure 6.10(d) and 6.10(e). The overlap causes about twice as many data page accesses for *rs* and nearly three times more for *rs-30*.

After dimension seven the cache is not sufficient to hold the complete {R}-Tree index anymore, therefore performance deteriorates and the {R}-Trees require $\approx$ 7.5 times more physical page reads than *ub/bub* in dimension nine, mainly caused by index page reads (Figure 6.10(f)). *ub/bub* work much better here and their cache requirements are much lower, as they are always traversing only one path during insertion.

Page Writes: The effects of reinsertion also become apparent when looking at the logical writes Figure 6.11(a), i.e., *rs-30* requires about 50% more logical writes. The actual physical writes correlate to the number of data pages and are similar for all indexes. Those are mainly caused by data pages writes, but with increasing index size also the *rs-30* starts to require more.

6.13.1.2 Real World Data

Additionally to artificial data, also the loading performance for real world data was benchmarked. The data are two dimensional points describing objects in a city in Poland, see Appendix B.2 for more details on the data. 144136 points corresponding to the caption texts of buildings are loaded. The page size is 2KB, which is a realistic page size for DBMS, but still relatively small for typical todays setups. The domain of the dimensions was in the range of $[4M, 6.5M]$.

Loading by random insertion was performed on the system described in Appendix A.1 and the following index variants have been tested:

Label	Index	$\mathbf{U_{min}}$	Other Parameters
ub =	UB-Tree	50%	
bub =	BUB-Tree	50%	
bub-100-20 =	BUB-Tree	0%	$R_{min} = 0\%$, $G^R_{min} = 20\%$
rs =	R*-Tree	40%	without reinsertion
rs-30 =	R*-Tree	40%	with reinsertion of 30%

Figure 6.12(a) shows the time required to load the complete data set. The R*-Tree variants require by an order of magnitude longer than the UB-Tree and BUB-Tree variants. *rs* needs 30 times longer and *rs-30* 81 times longer. Figure 6.12(b) reveals that this is not caused by the logical page accesses as in case of artificial data. Although the accesses are higher they do not explain the elapsed times.

There is one major reason causing this in comparison to the artificial data set: the increased page capacity of 2KB! 253 tuples are now fitting onto a data page and 101 index entries on a index page. {R}-Trees do not sort the entries on a page, which has the following consequences:

- The R*-Tree always has to check the complete page content when searching for an entry.
- The R*-Tree page split algorithm becomes more costly when allowing a page utilization below 50% due to more partitionings than need to be considered. With increasing page capacity also the number of possible partitionings increase.

In contrast to this, the UB-Tree and BUB-Tree store entries in address order and thus can apply an efficient binary search.

Reinsertion nearly doubles the page accesses for *rs-30*, while it increases the required time by ≈ 2.5. Both BUB-Tree variants require only a fraction longer than the UB-Tree.

Figure 6.12(c) reveals that the advantage of the BUB-Tree due to shorter index entries is now reduced as the domain of the dimensions is much larger now than it was for the artificial data. *ub* is and will always be superior as it only stores separators, but not bounding values. However, the BUB-Tree has not half the index entries, but 60% as there are not only the keys, but also the links to child nodes.

Compression of index pages is depicted in Figure 6.12(d). With values distributed on a bigger domain the R*-Tree cannot maintain the good compression that was observed for the artificial data.

The overlap of MBBs is the reason why {R}-Trees cannot provide performance guarantees. While it is possible to create pathological worst case distributions of MBBs, the worst case usually will not happen in practice. Figure 6.12(e) shows the number of MBBs at the last index level which overlap a point in the universes w.r. to the number of overlaps. The worst case is where twelve MBBs of the *rs-30* overlap one point. Nevertheless, the majority of points is only covered by one MBB and the number of occurrences of more overlaps decreases quickly. Reinsertion has the opposite effect here, causing more overlaps than without. This is partly caused by *rs-30* having fewer data pages and selection of wrong insertion paths.

Finally, Figure 6.12(f) depicts the coverage of the indexes. The R*-Trees have a coverage of $\approx 8\%$ while *bub-100-20* only reaches 50%. MBBs prove to be more flexible here, but the drawback is their overlap which is higher than the coverage, but not above 100% as for higher dimensions with the artificial data.

The BUB-Tree cannot do better here. Some regions stay bloated when created in the beginning and if no more insertions to them occur later. Only a reorganization can help here. Figure 6.13 shows the different partitionings. The majority of the BUB-Tree coverage stem from a few bloated regions. Therefore, decreasing the threshold for the gap size does not enhance the coverage, as it will only create further page splits in the densely populated areas.

As we will see in the next section this can be avoided during bulk loading and thus it can also be fixed for random insertion by allowing a reorganization, which can be triggered when the coverage exceeds a user specified maximum coverage.

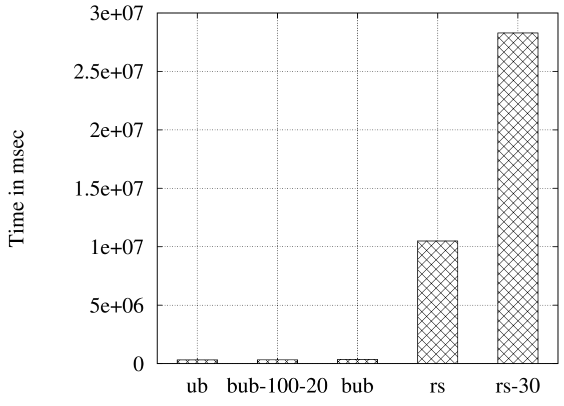
6.13.2 Bulk Loading

For bulk loading *rs* we use the R-Tree packing from [KF93]. Data is sorted according to the Hilbert-curve and then packed in that order into R-Tree-nodes of the designed page utilization. As a variant we also used the Z-curve, as it is more efficient to calculate addresses for it as we have seen before in Section 5.3.1.1.

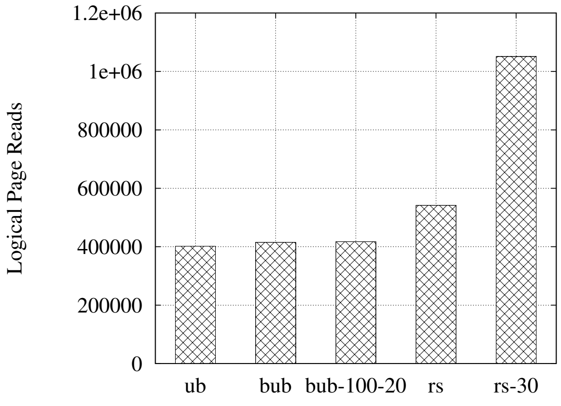
During our experiments we observed an interesting effect when inserting data into an R*-Tree which was previously sorted according to a SFC. Reinsertion works very well here and actually reduces the overlap to a large extent. Further, loading time and page accesses decrease in comparison to truly random insertion. This is caused by the locality of the insertions and thus causes to split of and reorganize the data only around the location of the current insertion. This can be seen as a specialized “bulk loading” technique, combining the clustering of a SFC with the split algorithm and reinsertion of R*-Trees.

Example 6.8 (R*-Tree MBBs caused by loading variants)

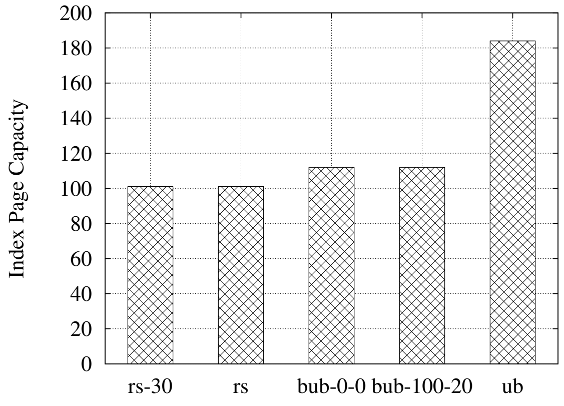
Figure 6.14 depicts the effects for a simple uniform data set. It shows the resulting MBBs partitioning at the last level of the index, i.e., for the data pages. The first row



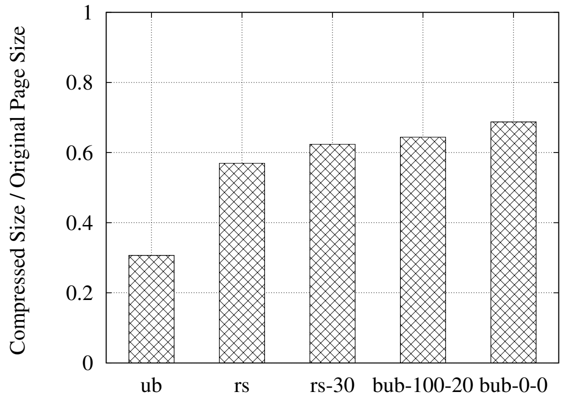
(a) Times



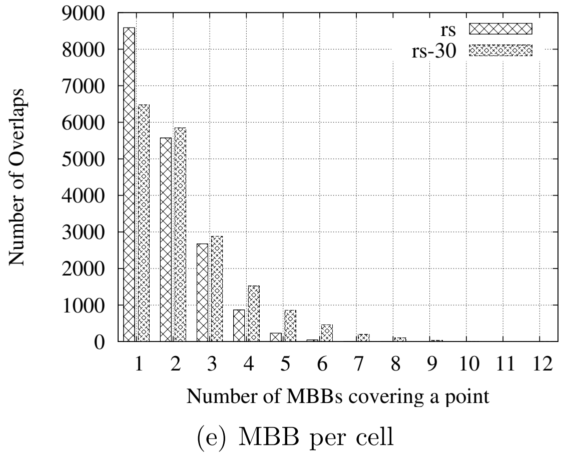
(b) Logical Reads



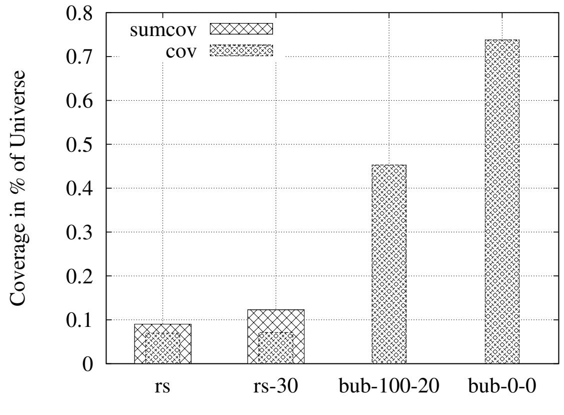
(c) Index Page Capacity



(d) Index Page Compression

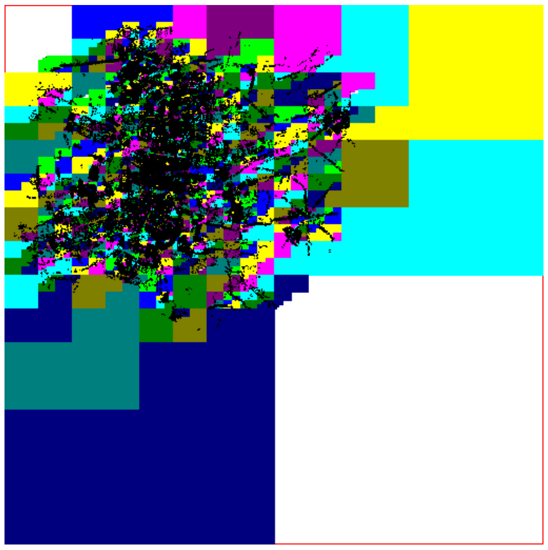


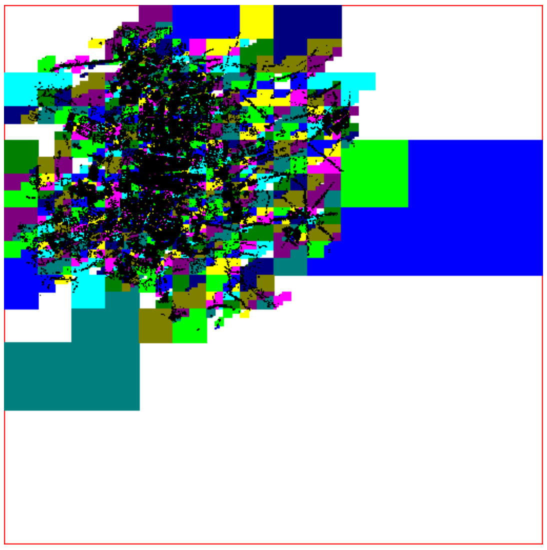
(e) MBB per cell

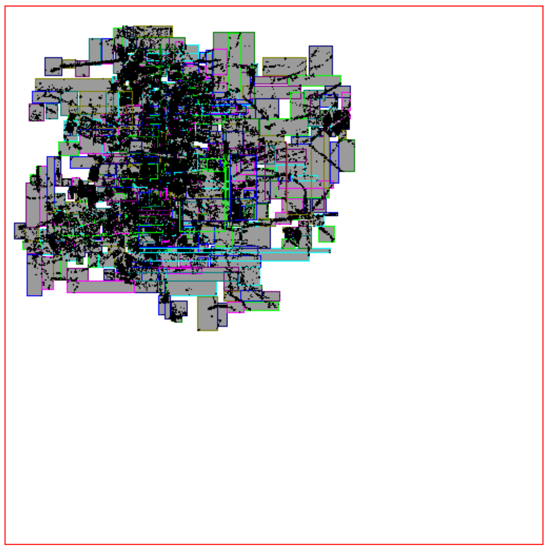


(f) Index Coverage

Figure 6.12: (GEO) Random Insertion: Overview


(a) *bub*


(b) *bub-100-20*


(c) *rs*

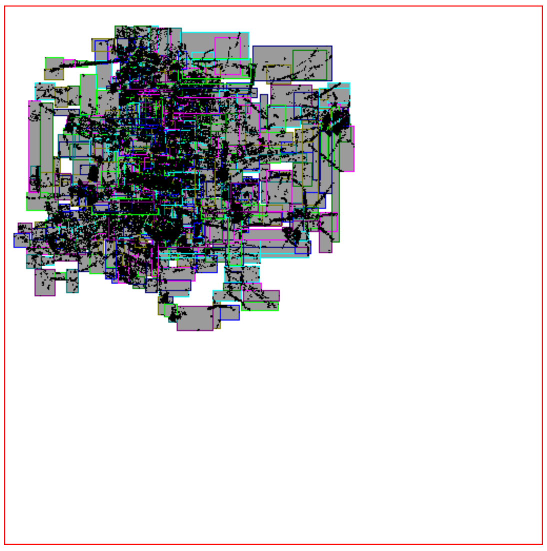

(d) *rs-30*

Figure 6.13: (GEO) Random Insertion: Page Writes

of figures corresponds to a R*-Tree without reinsertion, the second to a R*-Tree with 30% reinsertion and the last row to bulk loaded R*-Trees. The MBBs are depicted as shaded boxes. Figure 6.14(f) shows the positive effects of reinsertion in Hilbert-order in comparison to the R*-Tree without reinsertion in Figure 6.14(a) for unsorted data. Considering only Hilbert-order, Figure 6.14(f) shows the positive effects of reinsertion in comparison to the R*-Tree without reinsertion in Figure 6.14(c). Figure 6.14(e) for the Z-order shows that the positive effects of reinsertion will not always work, i.e., in comparison the R*-Tree without reinsertion in Figure 6.14(b). Inspecting the higher index levels of *rs-30-rand-z* reveals that due to reinsertion suboptimal paths have been chosen for reinsertions, causing overlap on the last level.

Figure 6.14(h) and 6.14(g) show bulk loaded R-Trees. The Z-curve causes here substantially more overlap due to its jumps. The pictures have the corresponding SFC-curve as overlay, thus showing the loading order and the tuples causing the MBBs. For example in Figure 6.14(g) there is one blue MBB covering nearly the complete x-axis domain. The major jump of the Z-curve causing this is denoted by an small arrow at the right side of the image. ◇

The experiments confirm that the Hilbert-curve is superior to the Z-curve w.r. to loading R*-Trees (see Example 6.8 on page 155). Due to shorter addresses, only two dimensions (in case of the real world data) and a quadratic universe, the Hilbert-curve address calculation is only ≈25% slower than the Z-curve addresses calculation here. In the following, the cost for preprocessing the input, i.e., the address calculation and sorting are not included in the loading times.

In the following we use the same data sets and setup as for the random insertion, but use bulk loading now.

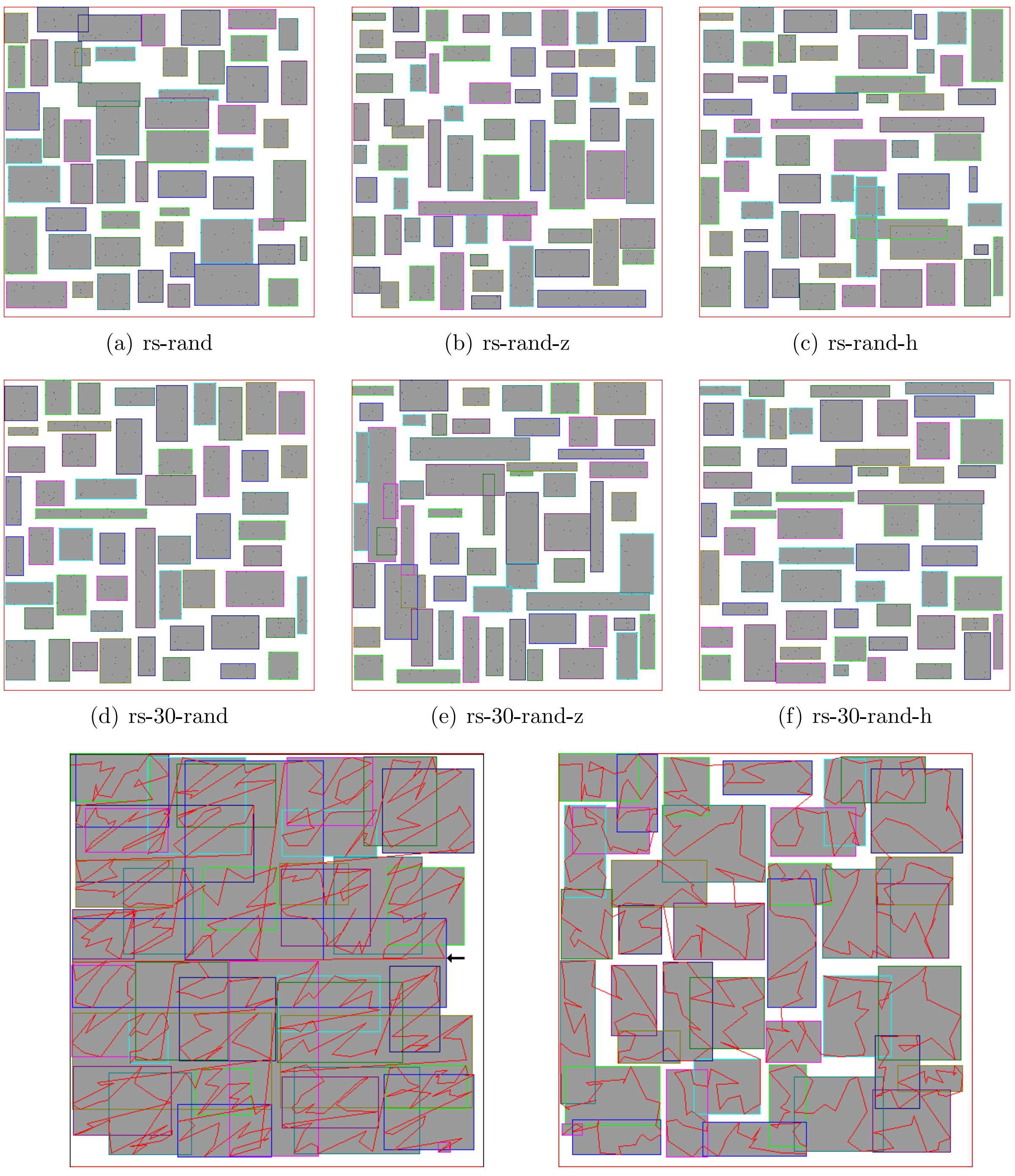
6.13.2.1 Uniform Data Distribution

First we want to take a look how the page split parameters of BUB-Trees affect the index size, coverage, creation time and page utilization. We have used the following naming schema, where *minRegion* is again the address prefix in percentage of the region size $\rho = \epsilon - \sigma$ of region $[\sigma, \epsilon]$ that must contain at least one 1-bit. The parameter *minGapSize* is the percentage of the gap w.r. to the region size ρ, and *multiSplitCount* the number of additional splits that might be allowed:

$$bub- < minRegion > - < minGapSize > - < multiSplitCount > \tag{6.12}$$

For the minimum region "size" r values between $[0\%, 100\%]$ are allowed, where 0% will prevent a split and 100% will enable a split regardless of the actual region size. The minimum region size G^R_{min} for r in percentage of the universe is:

$$G^R_{min} = 2^{\lfloor -r \cdot \sum_{i=1}^{d} \lceil \log_2 |D_i| \rceil \rfloor} \tag{6.13}$$



rs =	without reinsertion	-rand =	random insertion	-z =	Z-order
rs-30 =	with 30% reinsertion	-bulk =	bulk insertion	-h =	Hilbert-order

Figure 6.14: R*-Tree MBBs caused by loading variants

For the example data we have $log_2 D_i = 14$, thus $r = 10\%$ corresponds to 2 bits, which is a region size of $\frac{1}{2^2} = 25\%$ w.r. to the universe size $|\Omega|$. With increasing dimensionality the minimum region size drops exponentially.

For *minGapSize* values between $[0\%, 50\%]$ are allowed, where 0% will always trigger a split at the biggest gap, regardless of the actual size of the gap. For *multisplit* values of $[0, \ldots]$ are allowed, i.e., 0 causes no additional split, 1 is one additional split, 2 allows two additional splits thus the average page utilization is $\frac{1}{2+1} = \frac{1}{3}$. The number of actual splits is limited by the minimum page utilization U_{min} which was set to 0%, i.e., in worst case only one tuple will be on each page.

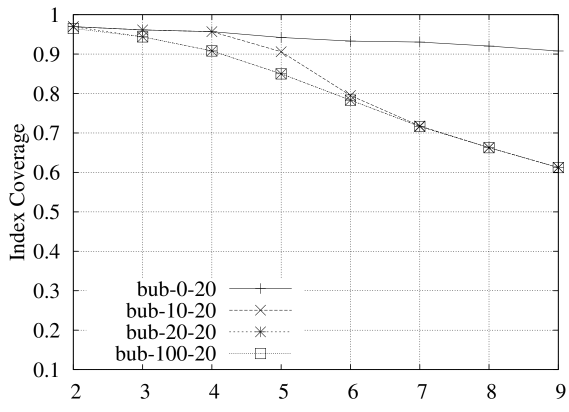
For the uniform data set we have performed extensive experiments varying all parameters separately while keeping the other parameters fixed. While there is not much dead space for uniformly distributed data in lower dimensions, dead space becomes more with increasing dimensionality. Also we use the uniformly distributed data set here, as it allows to understand effects more easily due to the known distribution. Figure 6.15 to 6.17 shows the results, where one column of figures refers to the same setup, but different measures and a row of figures to the same measure in different setups.

Where there was no significant difference between variations within a figure, we have omitted intermediate plots and only keep the extremes. For example, in Figure 6.16(b) the steps 40%-90% have been omitted as they are approximately equal to the 30% resp. 100% case. Labels are ordered according to the order of curves in the coverage plot (Figure 6.15(a) to 6.17(a)).

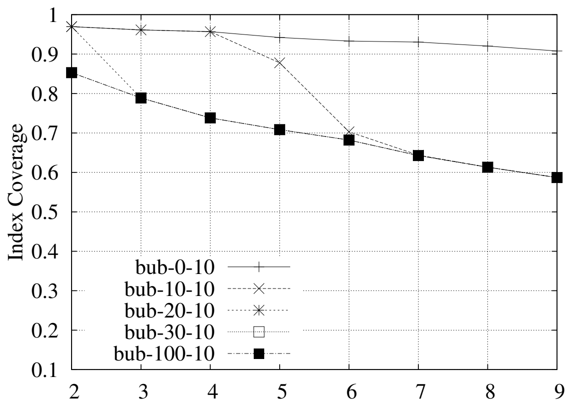
***bub-x-20*:** The first setup varies the minimum region size allowing a split for a fixed gap size of 20% w.r. to the region size (left column of Figure 6.15 and 6.16). We discuss the first column of figures. Already with a normal split the coverage decreases with increasing dimensionality due to the exponential increase of the universe size (*bub-0-20* in Figure 6.15(a)). For $r = 10\%$ the coverage decreases starting with dimension five. For higher r it decreases starting with dimension 2. For dimension nine it is 30% smaller than without best split ($r = 0$). At the same time the number of pages increases by 60% (Figure 6.15(c)/Figure 6.16(c)) and data page utilization stagnates at 60%, while the index page utilization increases. The creation time increases by 50% in comparison to the version without best split.

***bub-x-10*:** This setup also varies the minimum region size, but with a fixed gap size of 10% (right column of Figure 6.15 and 6.16). The smaller gap size allows more best splits, thus leading to an earlier decrease in index converge due to more of data pages. Consequently, also the data page utilization decreases earlier. At dimensions higher than six, the differences become smaller, leaving only a minor advantage.

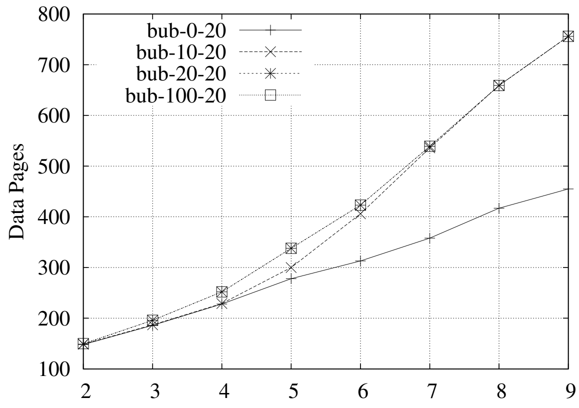
***bub-100-x*:** This setup also varies the minimum gap size while allowing a split regardless of the region size (left column of Figure 6.17 and 6.18). Only at a gap size lower than 50%



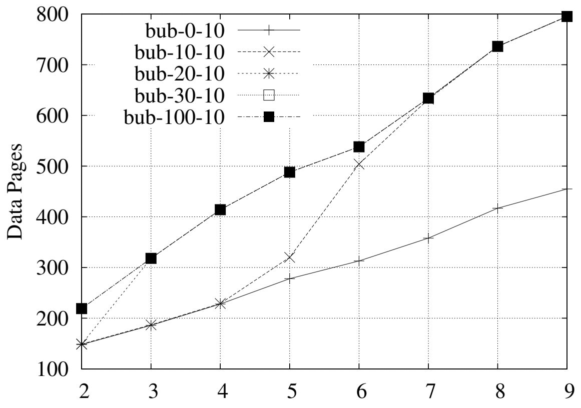
(a) Icov: bub-x-20



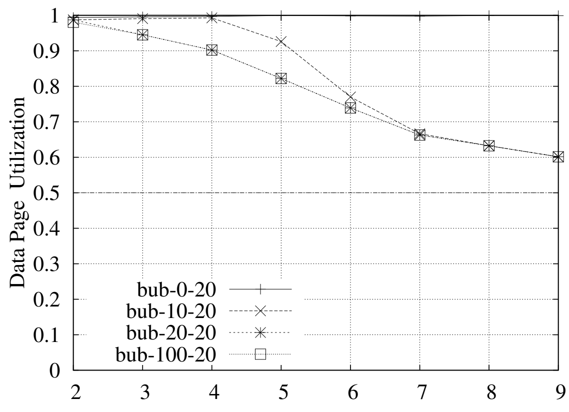
(b) Icov: bub-x-10



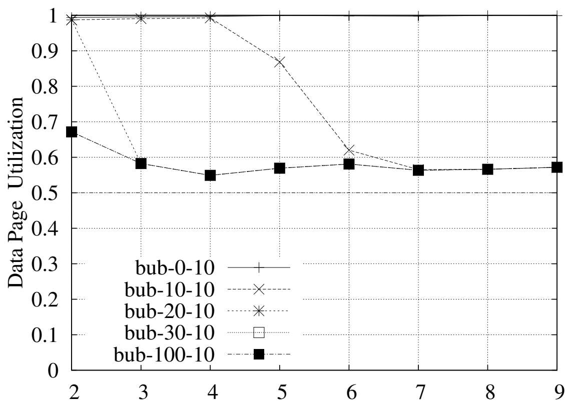
(c) Dp: bub-x-20



(d) Dp: bub-x-10

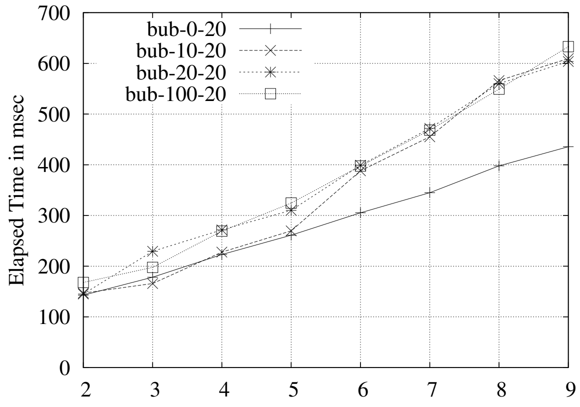


(e) Du: bub-x-20

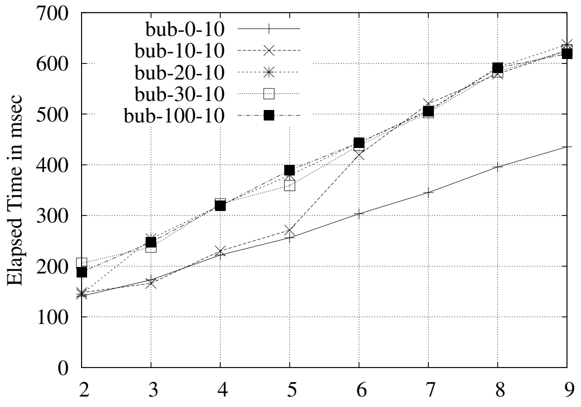


(f) Du: bub-x-10

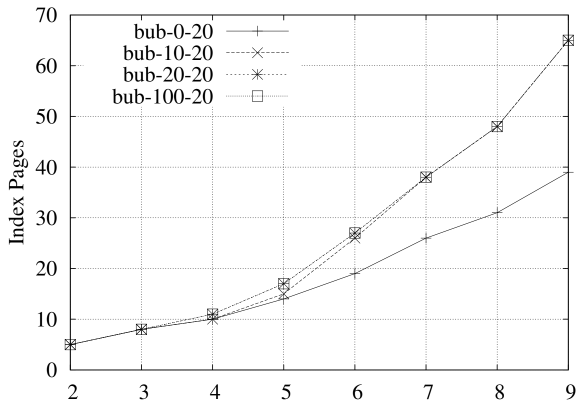
Figure 6.15: (Uniform) BUB-Tree loading parameters



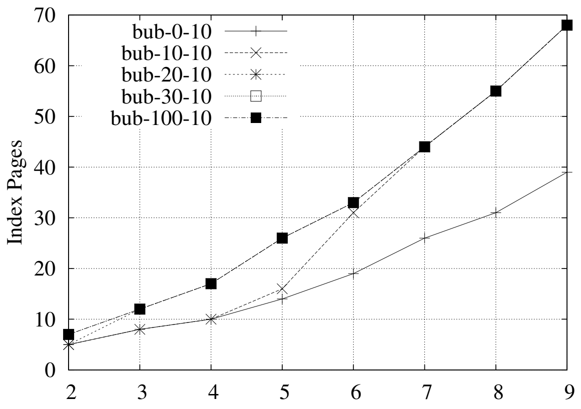
(a) time: bub-x-20



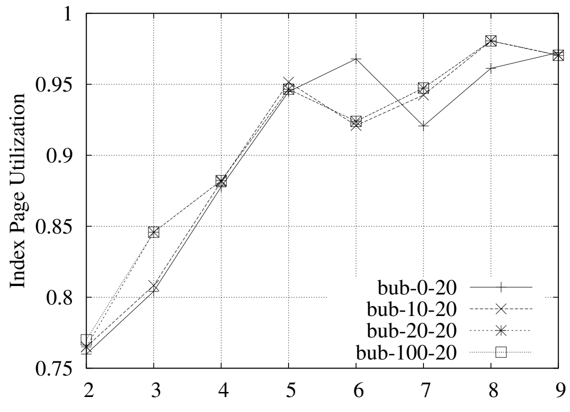
(b) time: bub-x-10



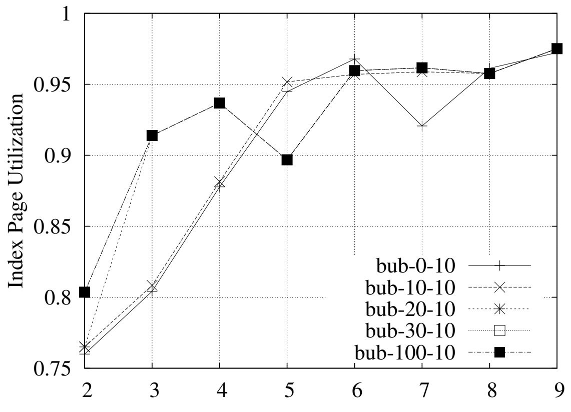
(c) Ip: bub-x-20



(d) Ip: bub-x-10

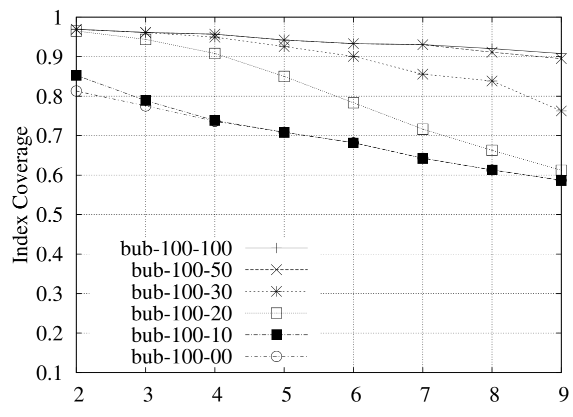


(e) Iu: bub-x-20

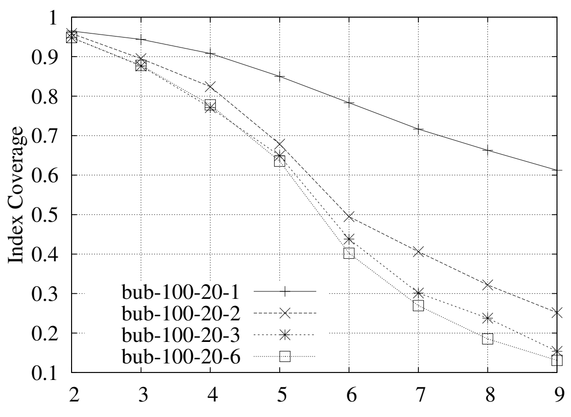


(f) Iu: bub-x-10

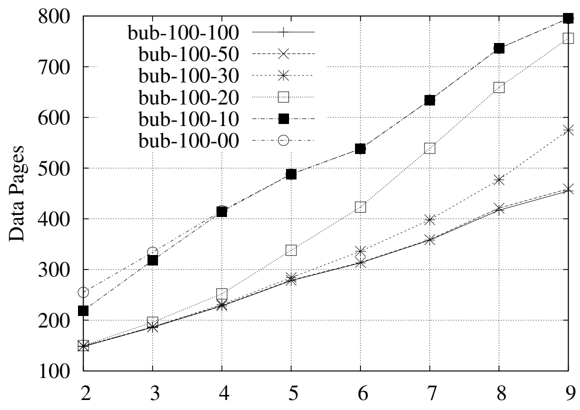
Figure 6.16: (Uniform) BUB-Tree loading parameters



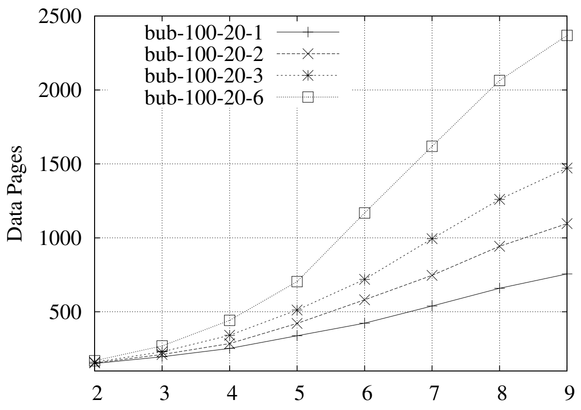
(a) Icov: bub-100-x



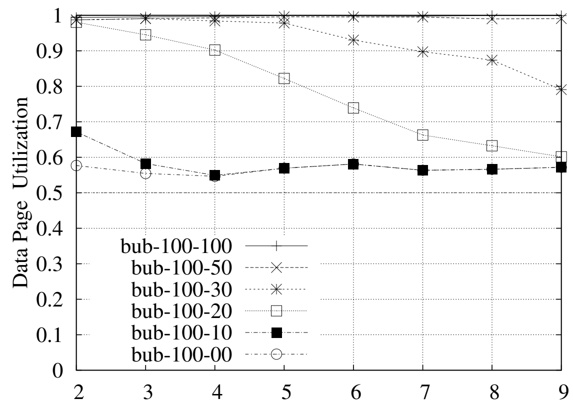
(b) Icov: bub-100-20-x



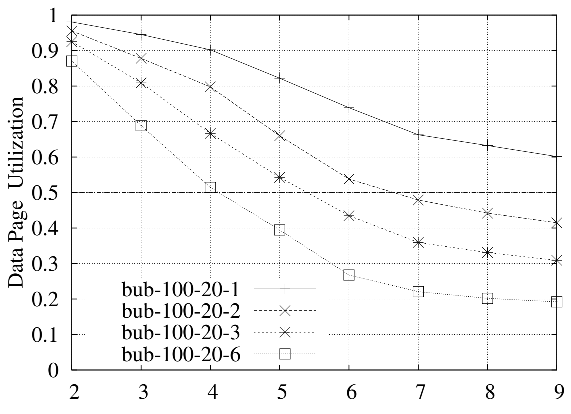
(c) Dp: bub-100-x



(d) Dp: bub-100-20-x

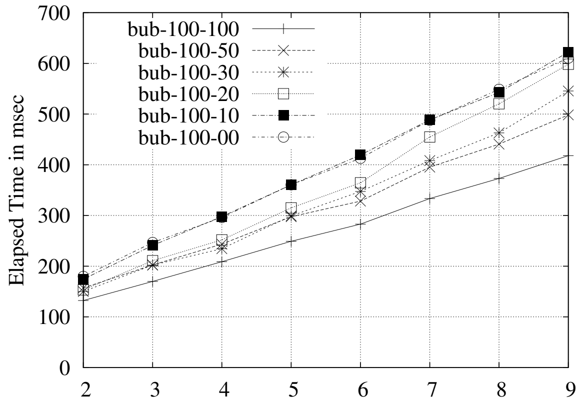


(e) Du: bub-100-x

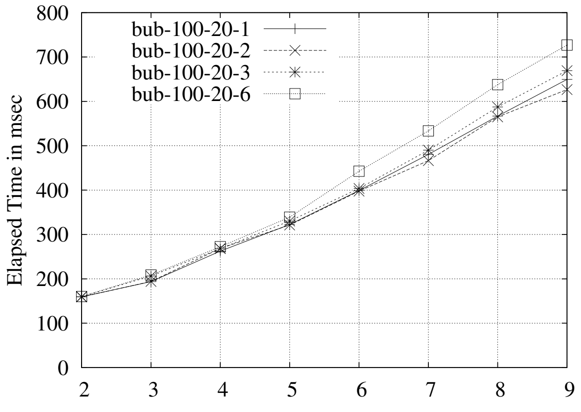


(f) Du: bub-100-20-x

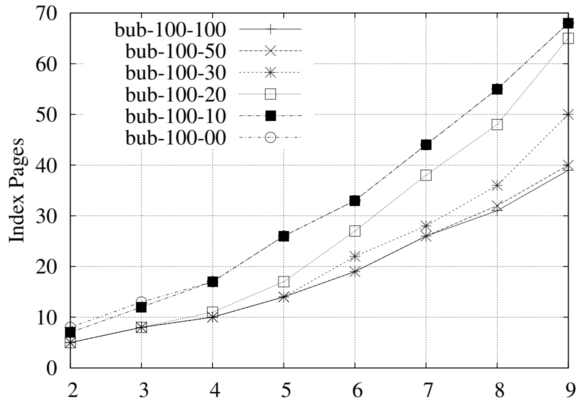
Figure 6.17: (Uniform) BUB-Tree loading parameters



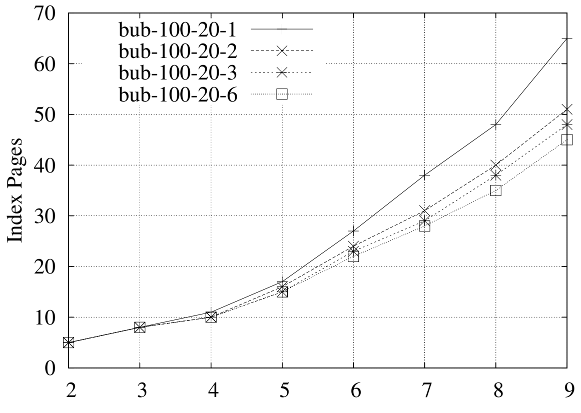
(a) time: bub-100-x



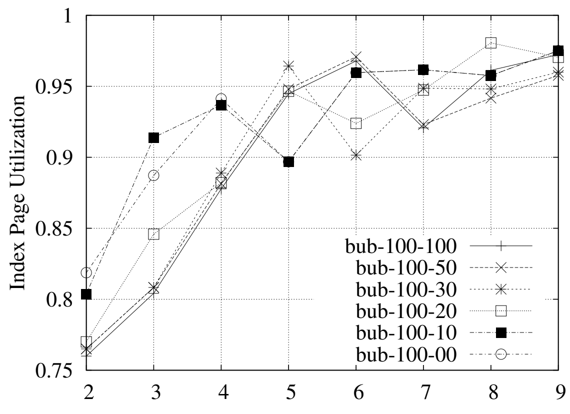
(b) time: bub-100-20-x



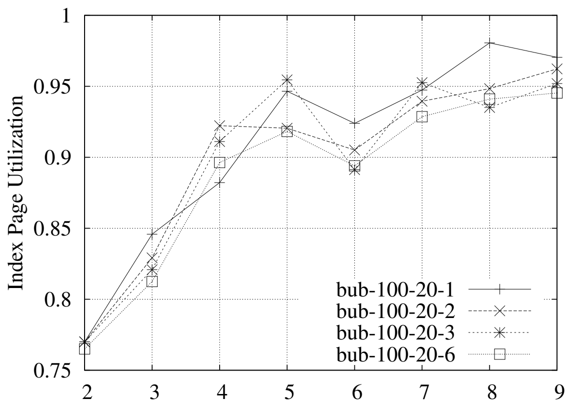
(c) Ip: bub-100-x



(d) Ip: bub-100-20-x

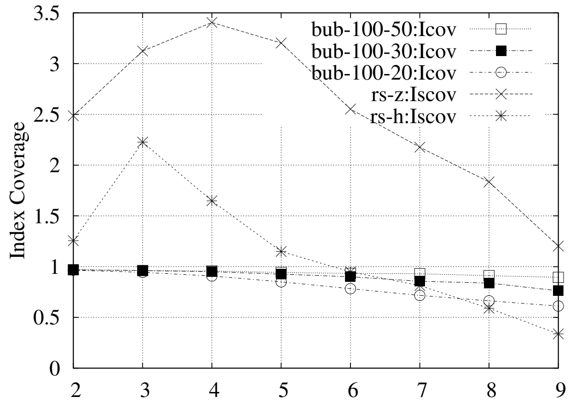


(e) Iu: bub-100-x

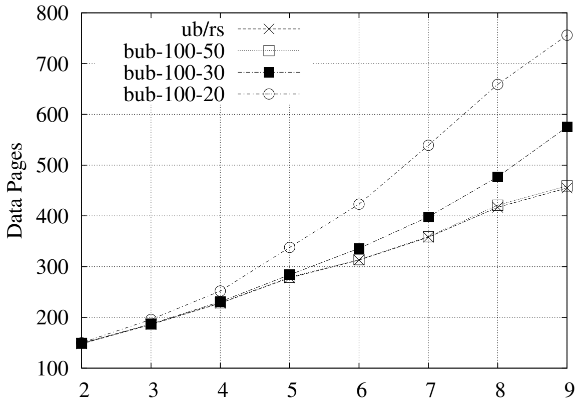


(f) Iu: bub-100-20-x

Figure 6.18: (Uniform) BUB-Tree loading parameters



(a) Index Coverage



(b) Number of Data Pages

Figure 6.19: (Uniform) Bulk Loading

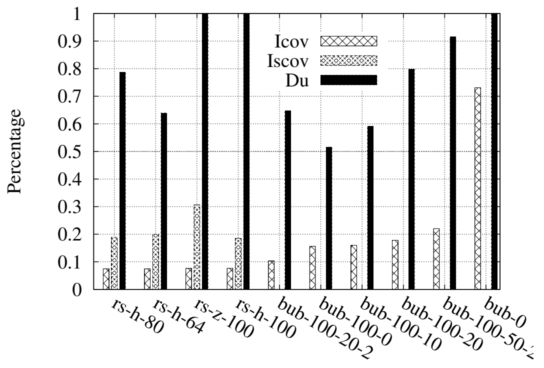
coverage improves. With a gap size of 20% it approximates the coverage of the case ignoring the gap size (*bub-100-0*). With a size of 10% it declines already at dimension two.

***bub-100-20-x*:** As there might be more than one gap worth a split to prune dead space we also allow multiple splits (right column of Figure 6.17 and 6.18). As we have set $U_{min} = 0$ any number up to the number of gaps would be possible. however as this deteriorates the page utilization we just allow up to six splits. This reduces the average page utilization to ≈20% (Figure 6.17(f)), but at the same time also the coverage was greatly reduced (Figure 6.17(b)).

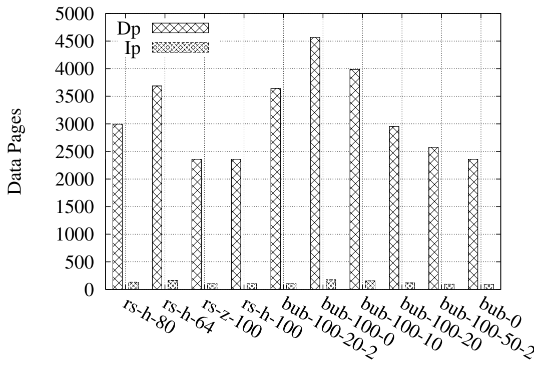
BUB-Tree Summary: While the data page utilization can be highly reduced during loading, further splits in the index part are performed only by the standard parameters and thus index page utilization remains high for all indexes.

Also loading times increase in comparison to loading without best split, still the differences between best split variants are small.

R*-Tree Summary: Figure 6.19 shows a comparison between index types. The coverage is in Figure 6.19(a). The Hilbert-curve packing for R*-Trees is clearly better than the Z-curve, i.e., the *sum-coverage* is lower and thus also fewer overlaps occur, in fact, starting with dimensions 6, it is even better than the BUB-Tree variants while preserving a 100% page utilization.



(a) Coverage and Page Utilization



(b) Index Size

Figure 6.20: (GEO) Bulk Loaded: Index Properties

6.13.2.2 Real World Data

With the skewed data distribution of real world data there are more chances of big gaps and thus to prune dead space. The page size was set to 512 byte. The coverage of the R*-Tree reduces to 7% and the sum-coverage drops to 18% for *rs-h* (Figure 6.20(a)). In comparison to Figure 6.13 on page 157, Figure 6.21 also shows a significant improvement of the BUB-Tree space partitioning.

With a slightly decreased page utilization of 92% the BUB-Tree *bub-100-50-2* reaches a coverage of 22%. For *bub-100-20-2* the coverage is decreased to 10% causing a drop of the data page utilization to 64%.

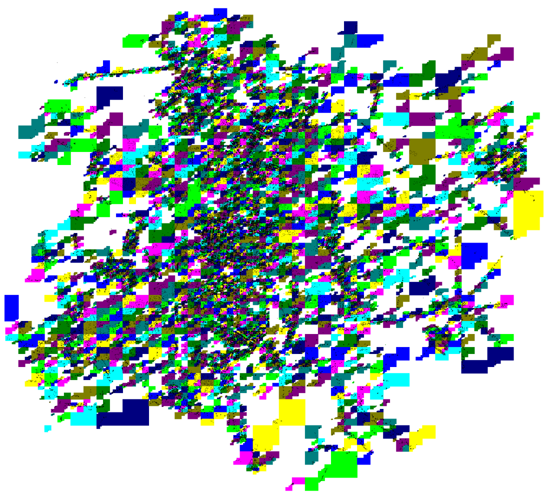
The reduced page utilization causes the creation of more pages, but may allow compacter MBBs. However, there is also a also a higher probability of more page accesses during query processing. Therefore, we also added variants of the R*-Tree with a page utilization of 64% resp. 80% corresponding to the page utilization of *bub-100-20-2* and *bub-100-20*, but the reduced page utilization has no significant impact on the coverage.

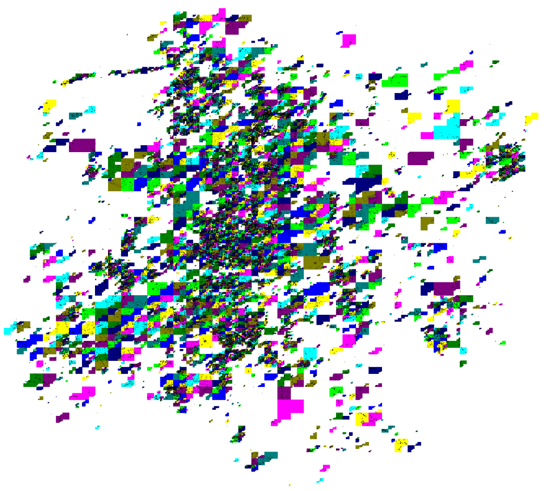
6.13.3 Query Procession

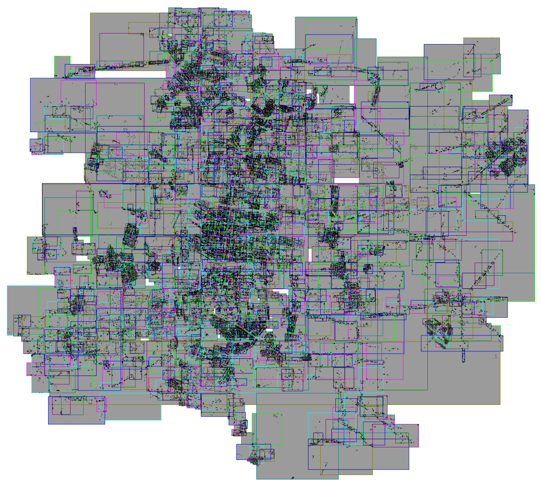
In the following we present the results of query processing. We have selected two different query types: Point and range queries. For each query type we have used the random and the bulk loaded indexes of both data sets, i.e., the uniform and the real world data set.

Further, we have divided each query set into queries with an empty result, thus they are probably in dead space and queries with a result, i.e., populated space queries.

Again, a cache size of 100 pages is used. Physical accesses have been ensured to take 4ms unless within a prefetched range of 10 pages. The cache has been invalidated before each query. The labels used for measures are again the same as before and as listed in Appendix C.


(a) *bub-100-20*


(b) *bub-100-20-2*


(c) *rs-z*

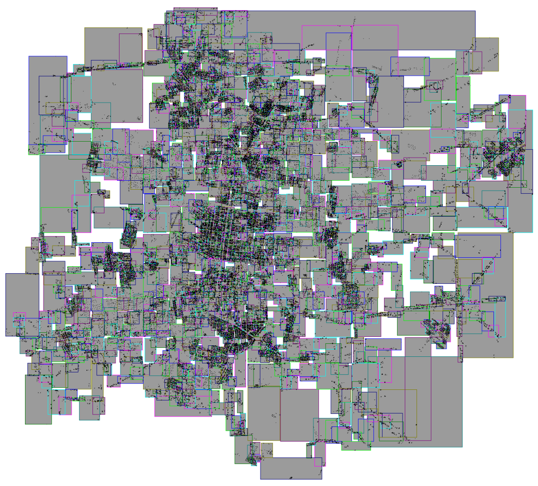

(d) *rs-h*

Figure 6.21: (GEO) Bulk Loaded: Partitioning
Zoomed into populated area.

6.13.4 Point Queries

Point queries are the fundamental primitive, required to insert, update and delete a point and thus are essential for good maintenance performance.

6.13.4.1 Uniform Data Set

We selected a set of ten queries, in order to avoid cluttering the plots. On the x-axis the last query of a dimension is marked by a label with the dimension number. Figure 6.22 on the facing page shows the results of running point queries on the uniform data set, which had no result. The UB-Tree has been omitted in the plots as its results are equal to the upper bound of the BUB-Tree.

The results for the indexes built by random insertion (left column of figures) reflect what has been already observed before. The R*-Tree suffers from its overlap which increases with dimensionality causing many index page accesses (Figure 6.22(c)). While it is able to avoid some data page access (Figure 6.22(e)) the dominating cost are the index page accesses. In contrast to this, the UB-Tree and BUB-Tree have a constant cost, i.e., one traversal down the tree. Sometimes the BUB-Tree can avoid data pages accesses, but not very frequently.

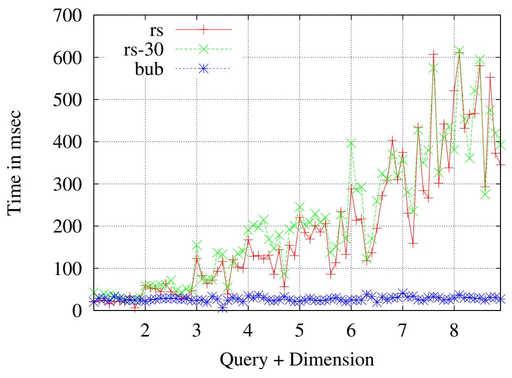
Running the same queries on a bulk loaded index mainly improves the performance of the R*-Tree variants. Here again the Hilbert-curve packed R*-Tree *rs-h* performs better, still the UB-Tree and BUB-Tree are better and the BUB-Tree is able to avoid more data page accesses (Figure 6.22(f)).

Additionally, we were also running a set of queries containing randomly selected tuples from the loaded data sets, i.e., queries with a result. For the R*-Tree indexes built by random insertion this even sightly increased the number of index page accesses. Also the data pages accesses increase to 4 in average. This is caused by the overlap of MBBs mainly occurring in populated space. For the bulk loaded indexes, the access to existing tuples slightly reduced the index page accesses for the R*-Trees. Data pages were reduced to an average of two pages for *rs-h* and four for *rs-z*. Again, the UB-Tree and BUB-Tree outperform this by accessing only one path in the tree, i.e., always a single data page.

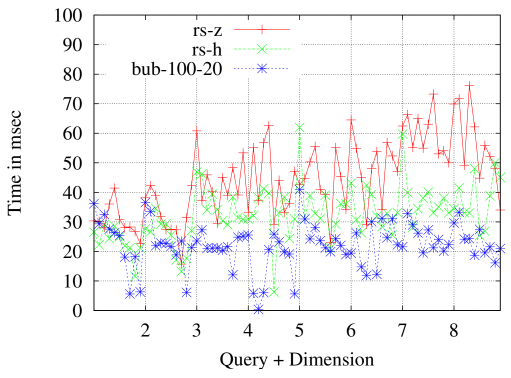
6.13.4.2 GEO Data Set

While the bulk loaded R*-Tree shows a considerable improvement in case of the uniform data set, it does not show the same advantages for the GEO data set w.r. to point queries without a result. There is one major reason: The dimensionality is only two, while significant benefits of bulk loading only showed up with dimensions higher than three.

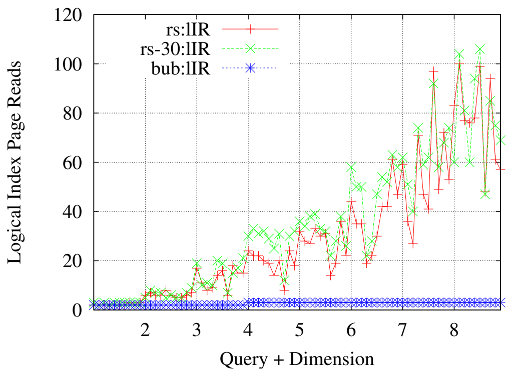
Figure 6.23(a) and 6.23(b) show the logical page accesses for 50 random point queries. If the R*-Tree is able to detect at the root page that the queried point does not exist, it causes only a single page access. While it is able to do this for some queries, others require inspection of multiple paths. With a R*-Tree tree height of 3 for the randomly loaded indexes resp. 4 pages for the bulk loaded indexes, this happens reasonably often (in more than 30%). In contrast, the BUB-Tree is always bound by its tree height 3 resp. 4, which



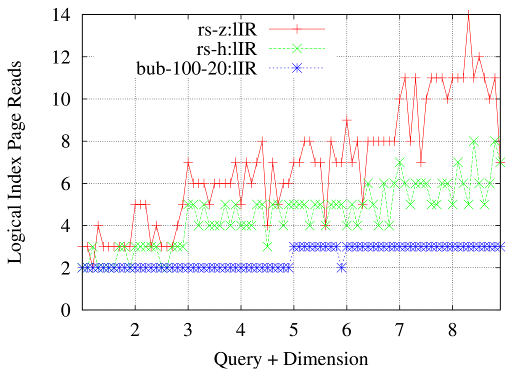
(a) (Random) time



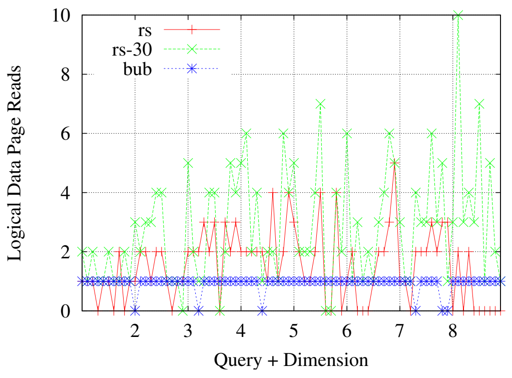
(b) (Bulk) time



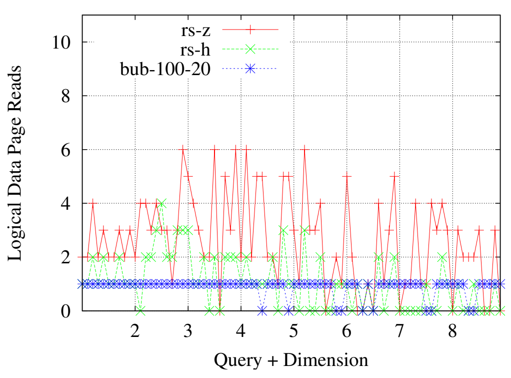
(c) (Random) lIR



(d) (Bulk) lIR

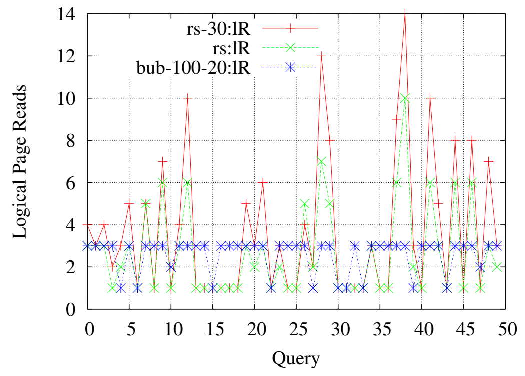


(e) (Random) lDR

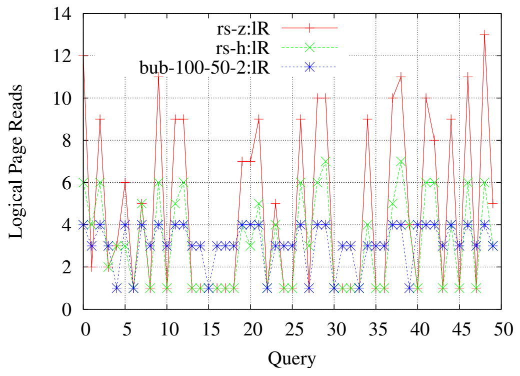


(f) (Bulk) lDR

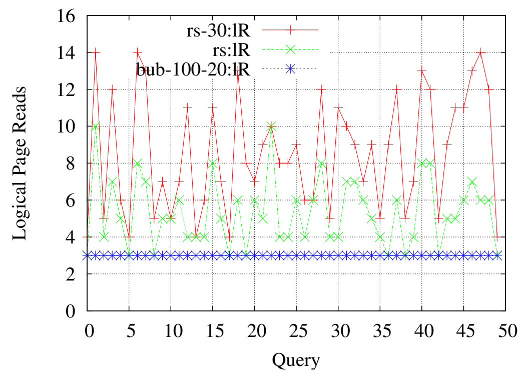
Figure 6.22: (Uniform) Point Query



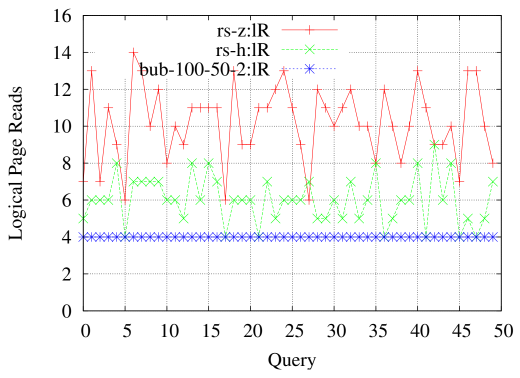
(a) (Random: non-existing) lR



(b) (Bulk: non-existing) lR



(c) (Random: existing) lR



(d) (Bulk: existing) lR

Figure 6.23: (Geo) Point Query

is the number of page accesses of the UB-Tree. Note that we have used different page sizes, i.e., 2048 for random insertion and 512 for bulk loading.

In case of querying existing points, bulk loading reduces the fluctuation in the number of accessed pages for the R*-Tree. Figure 6.23(c) and 6.23(d) show only the logical page accesses.

6.13.4.3 Summary on Point Queries

The results clearly show the advantage of the UB-Tree and BUB-Tree for point queries and thus for maintenance operations, i.e., insertion, update and delete. Further, the BUB-Tree reduces the number of page accesses where dead space was not covered by the index. While R*-Trees are able to prune more point searches at earlier levels of the index than the BUB-Tree, they also have higher number of queries requiring significantly more page accesses. Thus, in average the R*-Tree performs worse and its performance is not predictable. Although, the theoretical worst case of reading all pages does not happen in practice, the missing worst case guarantees must be considered as harmful.

6.13.5 Random Range Queries

While point queries are the measure for maintenance performance, range queries usually make up the majority of the query workload for multidimensional index structures. The actual shape of the range restriction has an important effect on the performance, i.e., "well-formed" ranges having approximately the same restriction in each dimension are usually handled better than partial match queries, which only restrict a few dimensions.

6.13.5.1 Uniform Data Set

For range queries without a result, the results are the same for all indexes. All have to access data pages and post-filtering them as there are not much chances to prune dead space. For each dimension a set of range queries with random location and size are executed including a mix of empty and none-empty results.

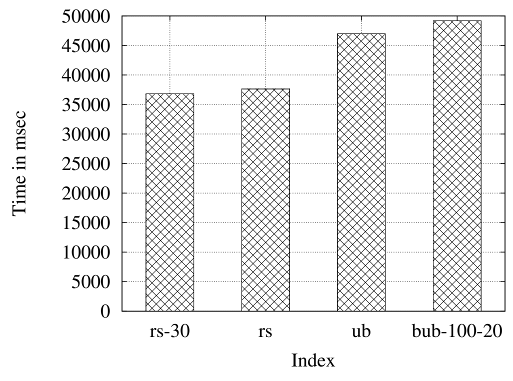
Indexes built by random insertion: The R*-Tree *rs* was performing better than the other indexes, where *ub* and *rs-30* were requiring 20% more and *bub* another 10% more time and page accesses. With higher dimensions the R*-Trees where showing the steep increases already seen for the point queries, while *ub* and *bub* are increasing only slightly. At dimension 7 to 9 the R*-Trees require four times more page accesses and time than the (B)UB-Tree.

Indexes built by bulk loading: Again, bulk loading helps the R*-Tree by reducing overlap in higher dimensions. The R*-Trees show a better performance on dead space queries, but worse on populated space due to overlap. In average the UB-Tree was performing 15% better than *bub-100-20* and *rs-h*, where the last two performed equally in average. The BUB-Tree suffers here from its larger index while not being able to prune a significant number of search paths.

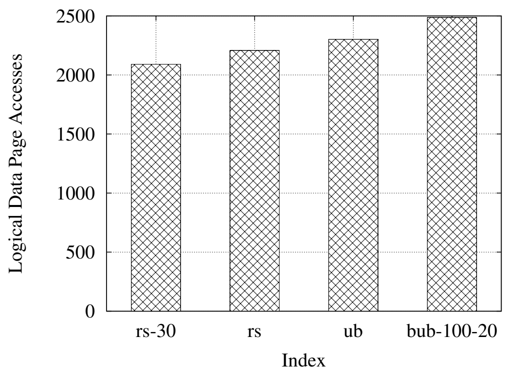
6.13.5.2 GEO Data Set

For the uniform data set, R*-Trees are only outperformed by the (B)UB-Tree with dimensions higher than three, therefore no significant advantage is to be expected for the GEO data set. Further, as bulk loaded indexes resulting in a page utilization lower than 100% there are likely to be more pages which will be accessed by a range query. Often a 100% page utilization is not desired as insertions to such an index immediately trigger page splits. It is only desired when the index is written only once and then will only be queried. Therefore, when comparing bulk loaded indexes, we will only compare those with the same page utilization.

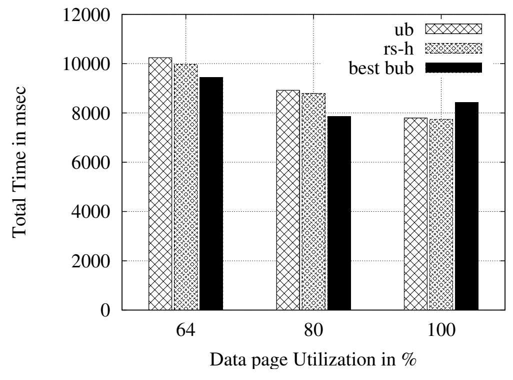
A set of random range queries with square like shape sizes ranging from small (point queries) to big (10% of the universe) are performed.



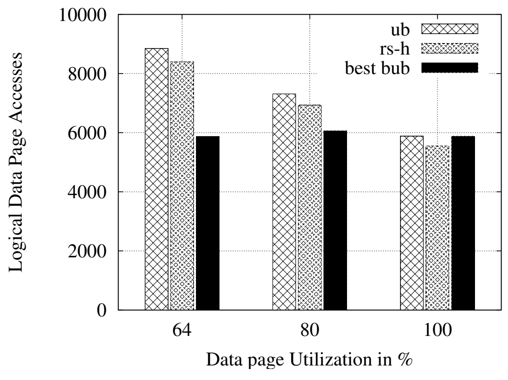
(a) (Rand) time



(b) (Rand) lDR



(c) (Bulk) time



(d) (Bulk) lDR

Figure 6.24: (Geo) Random Range Queries

Indexes built by random insertion: The R*-Tree *rs-30* is actually performing best (Figure 6.24(a) and 6.24(b)). Its MBBs fit well to the query shape. While for empty and smaller (below 10000) result sets the indexes were performing similar, with larger query boxes the UB-Tree and even more the BUB-Tree were causing additional page accesses. This is caused by the shape of Z-regions which do not align so well to the query shape like MBBs do. Thus, at the boundary of the range query they are causing more page accesses.

Indexes built by bulk loading: With bulk loaded indexes, the results are different. The BUB-Tree *bub-100-20-2* with a page utilization of 64% was performing better than the UB-Tree and the R*-Tree *rs-h* (see Figure 6.24(c)). The logical data page accesses have been reduced by more than 25% (see Figure 6.24(d)). With a page utilization of 80% *bub-100-50-2* was still performing better than *ub* and *rs-h*. However, for page utilization of 100%, the BUB-Tree was slower than the UB-Tree due to its larger index and *rs-h* was a little bit faster than *ub*.

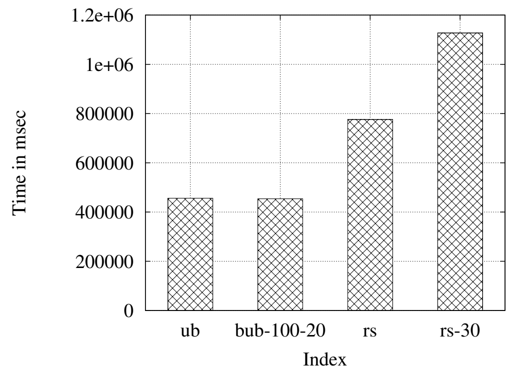
6.13.6 Street Section Queries

While the random range queries where artificial and well formed, we were also performing a set of more skewed real world queries: The bounding boxes of the street sections (Figure B.10(b) in Appendix B.2). This corresponds to the real world query: "Give me all buildings along a street segment". In fact, 73% of the queries were returning empty results due to the placement of building captions besides the streets and the alignment of most streets to south-north resp. east-west.

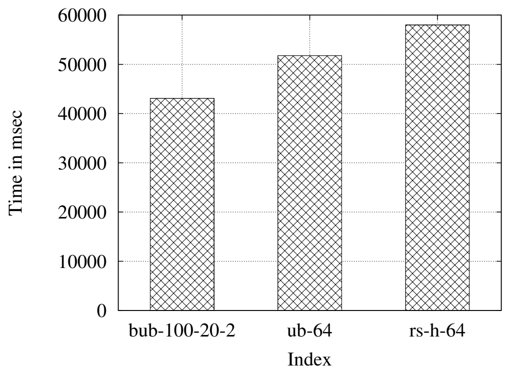
Indexes built by random insertion: UB-Tree and BUB-Tree were causing only half the data page accesses (left column of figures in Figure 6.25) in comparison to *rs-30* and they also outperformed *rs* by ≈30%. The UB-Tree and BUB-Tree were performing similar with BUB-Tree having some more page accesses. This is as expected as we are querying mainly populated space, i.e., the areas around streets. The elapsed time correlates directly to the number of page accesses.

Indexes built by bulk loading: The BUB-Tree is able to show its potential (right column of figures in Figure 6.25). It is causing 33% fewer data pages accesses. Also fewer index page accesses than the UB-Tree were required, even while having a smaller page capacity. There are two effects to mention here, query boxes are usually small and long due to representing street segments having a high probability of being aligned to south-north resp. east-west. The BUB-Tree is able to prune more dead space here, which is covered by MBBs in case of the R*-Tree.

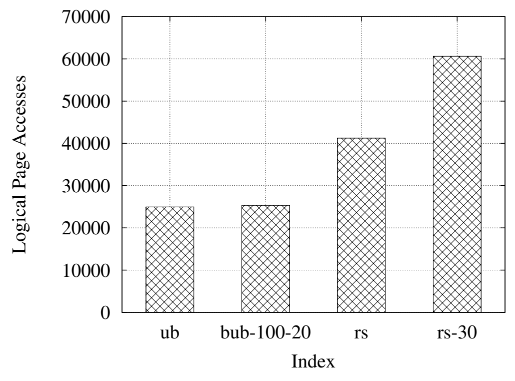
As we are cleaning the caches before each query, times correlate highly to the number of logical page accesses, i.e., index and data page accesses. With caching enabled between queries times correlate to the data page accesses, since the index gets nearly completely cached.



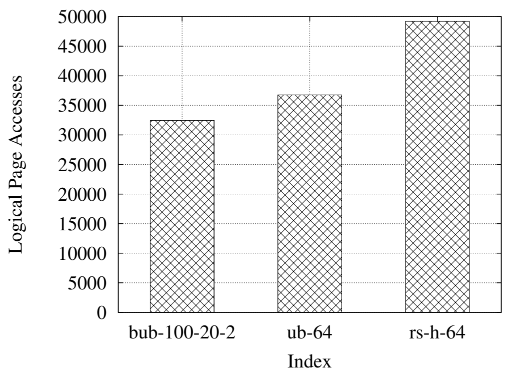
(a) (Rand) time



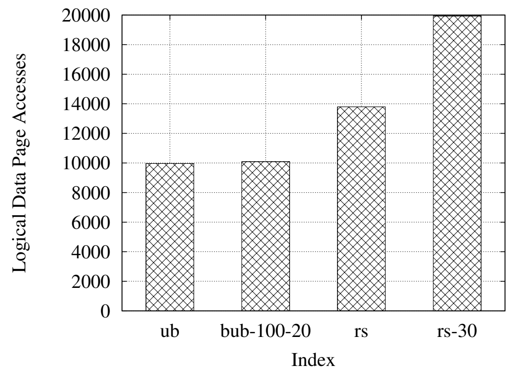
(b) (Bulk) time



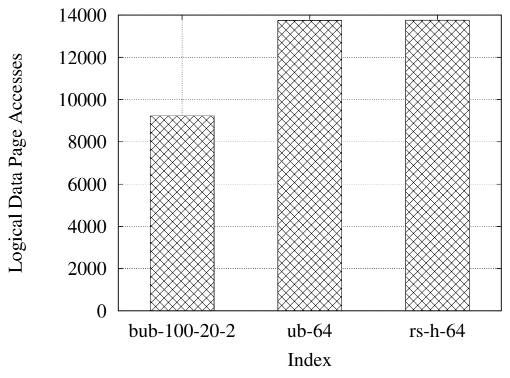
(c) (Rand) lR



(d) (Bulk) lR



(e) (Rand) lDR



(f) (Bulk) lDR

Figure 6.25: (Geo) Street Section Queries

The results for the street section queries on indexes built by random insertion are different in comparison to the artificially generated range queries. This also shows, how selecting the benchmark can influence the results and thus the rating of indexes.

6.14 Summary

We have presented a novel variant of the UB-Tree, called the *Bounding UB-Tree*. It is designed to handle dead space by pruning search paths to it already at the index level. Efficient algorithms for building, maintaining, and querying of BUB-Trees have been presented.

Summarizing the results of the measurements, the BUB-Tree provides excellent performance for maintenance operations, i.e., it inherits the worst case guarantees of UB-Tree resp. B-Trees. Thus it is suitable for dynamic applications, especially as with the presented online reorganization algorithms its performance can be further tuned while only slightly affecting parallel query processing. For bulk loaded data it shows a superior performance in comparison to other index types with the same page utilization.

R*-Trees can provide reasonable performance for queries covering dead space and when being built by bulk loading also on populated space. But they are not able to provide worst case guarantees, maintain good performance after random insertions, and the performance required for dynamic applications. While the UB-Tree already provides better maintenance and sometimes query performance, it cannot handle dead space due to its design. The BUB-Tree closes this gap while inheriting the good worst case guarantees for the basic operations from the UB-Tree.

Chapter 7

Management of Spatial Objects

Conventional RDBMSs have been designed to handle point data, i.e., one object is a row in a table and corresponds to a point. In contrast to this, spatial objects consist of an arbitrary set of points. Usually, the points are connected to each other defining a contiguous area. Special purpose DBMSs have filled the niche for efficient management of spatial data at the time where RDBMSs were not providing support for spatial objects.

However, advanced DBMS applications have set the demand for processing both, relational point as well as spatial objects in the same application and queries. Thus RDBMS vendors added support for spatial objects. For example, Oracle has added support for quad-trees and R-Trees and accompanying new data types for polygons as well as spatial query types.

In comparison to point data, spatial data and spatial queries are inherently more complex:

- Spatial objects in a given universe can be points, closed/open polygons, sets of points, lines, etc., all with a different description and varying storage requirements.
- There are more relations of objects to each other.

Therefore, virtually all approaches for indexing spatial objects use some kind of object abstraction or space partitioning. The most prominent abstraction is the approximation of objects by their minimum bounding box (MBB), the minimum axis parallel subcube containing the object.

The spatial index will only deliver a candidate set for a query, which usually must be post-filtered by evaluating spatial predicates on the candidates. In this chapter we focus on the step of finding the candidates. As evaluating spatial predicates is usually costly, the candidate set should be as small as possible, i.e., in best case equal to the actual result.

The management and processing of intervals is a special case of extended object handling with growing demand in various application areas.

Applications requiring interval matching and management include:

Temporal Databases: Standard databases store only the current version of the data – a snap shot, but with temporal databases we get the history of data and we have to deal with transaction and valid time ranges. [SOL94]

Quality Classes: Products like CPUs, cathode ray tubes and others are produced on the same assembly line and have to be classified according to some quality criteria (e.g., clock rate of the CPU). Again quality criteria may be expressed by a set of intervals and finding the right quality class is interval matching.

Personalization: Personalizing [KKaS97] a web-page requires the user to specify his profile. The process of personalization is now finding system profiles which map best to the user profile. Profiles can be expressed as sets of intervals, e.g., over accessed files. The solution is now based on finding the best system profile by intersecting them with the user profile.

Fuzzy Logic/Matching: The application area of fuzzy logic, resp. matching [Kau91] can also be mapped to interval matching. Fuzzy matching does not correspond to a point matching problem, but an interval matching where the interval width determines the fuzziness of the match.

Spatial Data: A Bounding box is only the set of intervals of each dimension. Further, spatial objects can be mapped to a set of intervals on a space filling curve. This can also be used for a more accurate approximation of objects and consequently results in better candidate sets than achieved by using the common technique of bounding boxes. However, this requires also additional storage and therefore it is reasonable only when dealing with objects which cannot be adequately approximated by bounding boxes [FR89, BKK99, KMP+01].

7.1 Related Work

[GG98] give a comprehensive introduction to spatial indexing and spatial index structures. Especially, the R*-Tree [BKS+90] has proven to be fairly efficient in comparisons to other techniques, but it cannot guarantee a worst case performance better than reading the whole index, even not for point queries.

7.1.1 RDBMS Implementations

While a vast number of different spatial index structures have been developed in the last decades, only grid approaches, the quad-tree and the R-Tree (and its variants) are noteworthy deployed in commercial DBMSs.

IBMs spatial extender [IBM02] uses an approach with grid cells. They use several layers from coarser to finer grids. For each grid a separate table stores which object MBBs cover what grid cell. Thus for overlapping objects there is one row for each object covering the cell. If an object covers more than four grid cells it is promoted to the next coarser grid level. Selecting the right size of grid cells is the crucial point here. If indexed objects change their extents over time the chosen cell size can become inaccurate. If objects become smaller than the finest cell size too many objects will be mapped to a single cell resulting in bigger candidate sets. In a similar way candidate set sizes will be too big if higher level grids are chosen too big in comparison to the next lower level, i.e., when promoted object map only to a single cell. Queries are handled by mapping the query shape to the set of covered grid cells on each level.

Microsofts TerraServer [BGS02] also maps objects to grid cells identified by the Z-address of their center. There are also grids for different resolutions. Using the Z-address ensures that physically nearby objects are also clustered nearby on disk. Queries are mapped to the set of grid cells covered by the query.

Oracle [Ora99] offers a quad-tree partitioning ordering sub cubes in Z-order. The number of levels of the partitioning has to be defined by the user. Choosing it wrong results in bad performance. Further, Oracle offers an R-Tree implemented by mapping the tree structure to a relation storing the child-father relations of R-Trees nodes. Due to this mapping the performance suffers [Nic01] as the node/page based access is not handled directly anymore.

Informix [Inf99] also offers an R-Tree implementation, which is well done and provides good performance in comparison to the R-Tree in Oracle.

Also most GIS DBMSs rely on the quad-tree or the R-Tree, e.g., SmallWorld [Sma02].

7.1.2 Interval Indexing

A huge number of different techniques for interval management has been proposed. [KS91, TCG$^+$93, Boz98, MTT00] give nice surveys on most of them. We only consider secondary storage index structures, since main memory data structures usually can not be mapped directly to existing DBMS technology, e.g., the external Segment-Tree [BG94] is a nontrivial mapping of the Segment-Tree.

Furthermore, we want to focus on indexing structures which can be used by exploiting the techniques provided by commercial RDBMSs, e.g., indexes like the B-Tree or UB-Tree. Therefore, we do not consider [SOL94, GLO$^+$96, KS91] which require indexing techniques not available in any commercial DBMS.

The use of regular B$^+$-Trees for indexing valid time intervals was suggested in [GLO$^+$96]. Here, the intervals are mapped to two-dimensional points with the same mapping function used for the TP-index [SOL94]. These two-dimensional points are mapped back to one-dimensional points (not intervals) by defining a total order among them using either horizontal, vertical, or diagonal sweep lines. B$^+$-trees are used to index these points after the final transformation. Temporal queries also go through these transformations. In this scheme, some specific temporal queries transform into range search queries for the B$^+$-trees, and can be efficiently evaluated. However, because of the transformations, many queries require multiple range search operations, and cannot be handled efficiently.

The SR-tree (Segment R-tree) [KS91] is a variant of the R-tree to index segments. Unlike the R-tree, the SR-tree keeps data also in the internal nodes. Any segment that spans any of the children of a node is kept in that node and is checked every time that node is visited in a search query. The SR-tree is a dynamic index, i.e., it allows deletions and insertions at any time. However, insertion and deletion algorithms may cause a high degree of overlap between the nodes. One should also mention that, although insertion and deletion times are logarithmically bound, they are relatively more expensive compared to index structures such as the B^+-/UB-Tree. The same drawbacks hold also for the R-Tree and the R^*-Tree. Moreover, no integrated R-Tree offers the excellent concurrency and maintenance properties of the B-/UB-Tree. Furthermore all the R-Tree variants integrated in commercial DBSMSs are secondary indexes and consequently they cannot perform as well as clustering indexes like the B-Tree or the UB-Tree.

The Relational Interval Tree (RI-Tree) [KPS00, KPS01, KMP+01] utilizes a virtual binary tree to partition the data space and group intervals to nodes of the binary tree. The conceptual structure of the RI-Tree is based on a virtual binary tree of height h which acts as a backbone over the range $[1, 2^h - 1]$ of potential interval bounds. Traversals are performed purely arithmetically by starting at the root value 2^{h-1} and proceeding in positive or negative steps of decreasing length 2^{h-i} , thus reaching any desired value of the data space in $O(h)$ time. This backbone structure is not materialized, and only the root value 2^h is stored persistently in a meta data tuple. For the relational storage of intervals, the nodes of the tree are used as artificial key values: Each interval is assigned to a *fork node*, i.e., the first intersected node when descending the tree from the root node down to the interval location.

An instance of the RI-Tree consists of two relational indexes which in an extensible indexing environment are at best managed as index-organized tables. These indexes then obey the relational schema *lowerIndex* $(node, start)$ and *upperIndex* $(node, end)$ and store the artificial *fork node* value, the *start* and *end* of the interval. Additionally, one can add an identifier or other attributes to the relation.

As any interval is represented by exactly one entry for each the lower and the upper bound, $O(n/b)$ disk blocks of size b suffice to store n intervals. For inserting or deleting intervals, the node values are determined arithmetically, and updating the indexes requires $O(\log bn)$ I/O operations per interval.

For processing intersection queries we collect the nodes of the binary tree which may contain intervals intersecting the query interval $[i_s, i_e]$. Theses nodes fall into three classes:

leftnodes: those nodes which are left to i_s and where we have to test $i_s \leq r_e$

rightnodes: those nodes which are to the right of i_e and where we have to test $r_s \leq i_e$

contained: and those nodes included by the query interval, i.e., $i_s \leq r_s \leq r_e \leq i_e$

The *leftnodes* and *rightnode* are stored in transient tables and a SQL query is executed that joins these with the base table while checking the predicates.

The RI-tree can be implemented by (procedural) SQL or within the application and does not assume any lower level interfaces. In particular, the built-in index structures of a DBMS are used as they are, and no intrusive augmentations or modifications of the database kernel are required.

7.1.3 Sub-Cube Indexing

[Gae95] describes the partitioning of spatial objects into a set of sub-cubes defined by the quad-tree partitioning. The sub-cubes are mapped to integer numbers and they are indexed by a B-Tree.

Sub-cubes can also be identified by their SFC-address. [FFS00] represents points (or regions) on SFC-curves, using a variable amount of bits and a length field.

7.1.4 Previous UB-Tree based solutions

For the UB-Tree [Bay97] proposes to store object IDs in a Z-region if the object covers this Z-region. As an object might be in multiple Z-regions its ID might be stored in several regions thus no storage requirements can be guaranteed in case of overlapping objects. Further, it requires access to Z-region boundaries for intersecting it with spatial objects and thus cannot be implemented on the application side on top of an existing UB-Tree implementation.

7.2 Mapping Methods

The most primitive spatial object is an one dimensional interval. The relation between two intervals are equal to many of the relations of spatial objects in general.

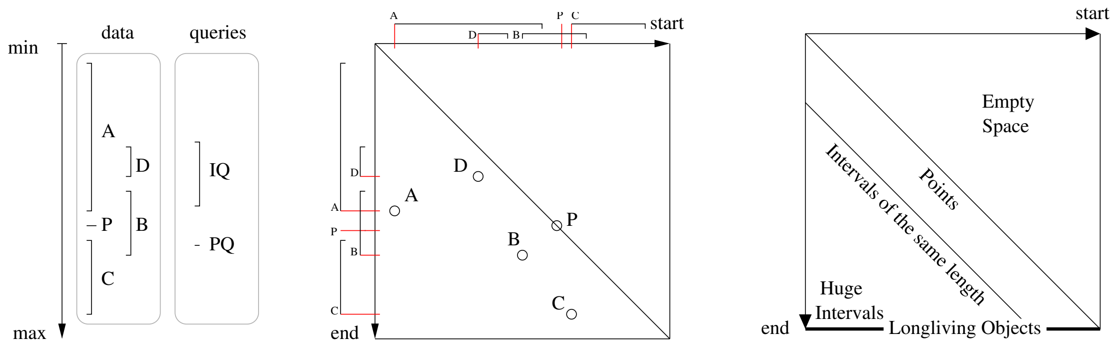
There are only three basic relations $(<, >, =)$ for two atomic values $a, b \in \mathbb{D}$, but much more for intervals. [AH85] defines thirteen relations for two time intervals $[s_1, e_2], [s_2, e_2]$ where $s_i, e_i \in \mathbb{D} \wedge s_i \leq e_i$. Besides *before* $e_1 < s_2$, *after* $e_2 < s_1$, and *equality* $s_1 = s_2 \wedge e_1 = e_2$, there are *containment* $s_1 \leq s_2 \wedge e_2 \leq e_1$, *enclosure* $s_2 \leq s_1 \wedge e_1 \leq e_2$, *intersects* $s_2 \leq e_1 \wedge s_1 \leq e_2$, *meets*, *met-by*, etc.

The *equality* and *intersection* query are regarded as the important primitives that must be supported by a spatial index.

In the following we discuss different approaches for handling spatial data and queries. All utilize the UB-Tree as a multidimensional point access method by transforming the spatial objects to points and the spatial queries to range queries on this new universe (referred to as *parameter space*).

7.2.1 Approximation by a MBB

One common solution to reduce the complexity of spatial objects is the approximation. It usually results in a fixed size representation which is indexed and guarantees that the result of queries is a superset of the actual result. An easy and efficiently to calculate approximation is the minimum bounding box (MBB). It is obtained by taking the minimum



(a) Native Space (b) Parameter Space (c) Properties

Figure 7.1: Endpoint Transformation

and maximum values of the object in each dimension. For a spatial object described as polygon of n points, this requires to inspect each point of the polygon once.

Simple geometric shapes can be considered as points in higher dimensional space called the *parameter space* [SK88, Ore90, GG98]. MBBs are simple, i.e., all MBBs for a given universe have the same size and structure. Thus, the transformation can be applied to them.

In the following we will use one dimensional intervals to depict the mapping of objects and queries. As MBBs are only a set of such intervals, one for each dimension, the mapping can be applied to each such interval without loss of generality.

MBBs are a good approximation for objects which are aligned to one or all of the axis, or which are convex and have a similar extent in all directions. However, for line segments not aligned and concave objects they are a bad approximation. The quality of the approximation of a MBB m for an object o, is the ratio of their covered space, i.e., $\frac{cov(m)}{cov(o)}$ where $cov(x)$ is defined as the set of all points in a discrete universe that intersect x or are contained within x.

7.2.1.1 End Point Transformation

The beginning and end of a MBB $[\vec{s}, \vec{e}]$ with $\vec{s} = (s_1, \ldots, s_d), \vec{e} = (e_1, \ldots, e_d) \in \Omega$ of dimensionality d are mapped to a point $(s_1, \ldots, s_d, e_1, \ldots, e_d)$ in $d \cdot 2$ space, simply by taking each component as individual dimension.

$$\begin{tikzcd}
\text{[start,end]} \arrow[r, "\text{End Point}", "\text{Transformation}"'] & \text{(start,end)} \\
\text{1D Interval} & \text{2D Point}
\end{tikzcd}$$

This doubles the dimensionality as depicted in Figure 7.1(b) for a simple example of one dimensional intervals in Figure 7.1(a).

This transformation results in the following properties of the parameter space, which are depicted for the two dimensional parameter space in Figure 7.1(c). When taking about

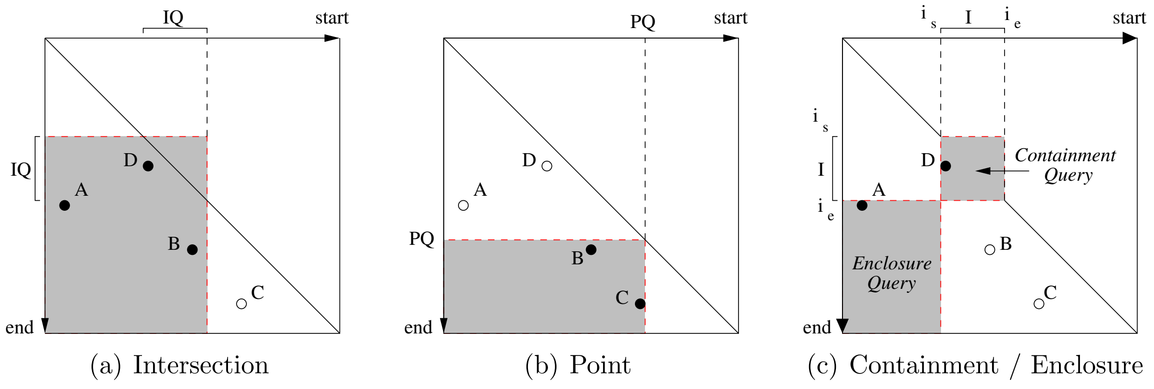


Figure 7.2: Endpoint transformation of Queries

the main diagonal we also refer to its equivalent in spaces with a dimensionality higher than two.

- The space above the main diagonal is empty, i.e., due to $s_i \leq e_i$ no points are mapped to this space.
- Points are placed on the main diagonal due to $\vec{s} = \vec{e}$.
- MBBs of the same length are mapped to a diagonal.
- The bigger a MBB gets the nearer it is to $(\vec{min}, \vec{max})$

In the context of temporal data (time intervals) the following can be applied:

- Long-living objects, i.e., time intervals where the end is not known, are mapped to $(start, max)$.

The *end-point* transformation maps intersection, containment, and enclosure query to a single range query in parameter space as depicted in Figure 7.2. A point query, i.e., all MBBs containing a given point, is a special case of intersection query, where the query shape is no MBB, but a point. In case of point queries the variable corner of the range query lies on the main diagonal. However, queries restricting only the size/extent of objects cannot be mapped to a range query, but they map to an parallelogram as depicted in Figure 7.3(a).

The exact mapping of queries is as follows:

Exact Match Query: EMQ$([i_s, i_e]) = \{[r_s, r_e] | r_s = i_s \wedge r_e = i_e\}$ is a point query in parameter space.

Intersection Query: IQ$([i_s, i_e]) = \{[r_s, r_e] | (r_s \leq i_e) \wedge (i_s \leq r_e)\}$ can be mapped to a query box $[(min, i_s), (i_e, max)]$ by adding the lower and upper bounds of the domain.

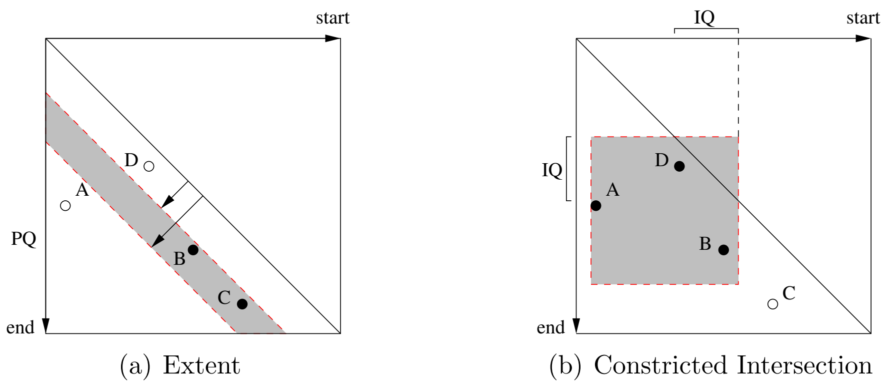


Figure 7.3: Extent match and Constricted Intersection Query

Point Query: PQ(p) = $\{[r_s, r_e] | r_s \leq p \leq r_e\}$ is actually a special case of IQ and results in the query box $[(min, p), (p, max)]$.

Containment Query: CQ($[i_s, i_e])$] = $\{[r_s, r_e] | (i_s \leq r_s \leq i_e) \wedge (i_s \leq r_e \leq i_e)\}$ maps simply to the query box $[(i_s, i_s), (i_e, i_e)]$.

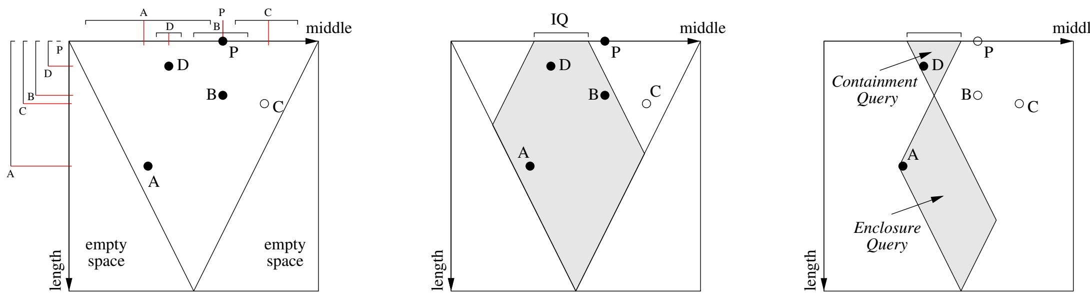
Enclosure Query: EQ($[i_s, i_e]$) = $\{[r_s, r_e] | (r_s \leq i_s \leq r_e) \wedge (r_s \leq i_e \leq r_e)\}$ can be simplified to $\{[r_s, r_e] | (r_s \leq i_s) \wedge (i_e \leq r_e)\}$ due to $r_s \leq r_e$ and $i_s \leq i_e$ and extended like IQ resulting in the query box $[(min, i_e), (i_s, max)]$.

Despite the conceptual elegance, the properties of the parameter space have been regarded as problematic for indexing [GG98]:

1. The parameter space doubles the number of dimensions, i.e., instead of indexing one object of type interval, one has to index a two dimensional space.
2. The data distribution in parameter space is highly skewed.

The data distribution in parameter space is highly skewed, i.e., start and end of intervals are usually highly correlated as there are more small objects and usually the length of the intervals in a data set is similar. Therefore, all data is below the main diagonal and usually distributed along some diagonals (intervals of similar length). Consequently, the used index has to handle skewed data distributions well. For example, the Grid-file [NHS84] cannot guarantee a good space utilization with this kind of data distribution and the R-tree and its variants cannot guarantee good query performance due to the overlap of bounding regions.

Due to these problems of parameter space, even a specialized index structure, the LSD-tree [HSW89] has been developed. The LSD-tree is an adaptive k-d-tree with a specialized paging algorithm. While efficient, this special paging strategy is obviously a major impediment for the integration of the LSD-tree into a general purpose database system ([GG98] page 28). The authors also propose to restrict the mapped range queries by taking the largest interval w.r. to a dimension into account. This reduces the size of the range query and thus the probability of intersecting pages (see Figure 7.3(b)).



(a) Parameter Space (b) Intersection (c) Enclosure/ Containment

Figure 7.4: Midpoint Transformation of Data and Queries

7.2.1.2 Mid Point + Lengths Transformation

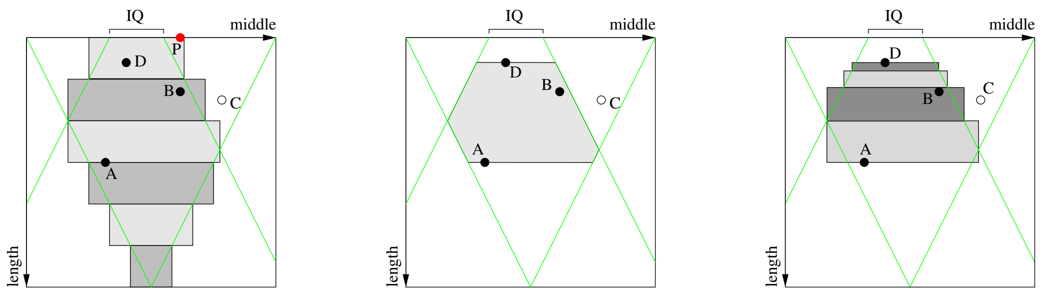
Instead of mapping the end-points one can also map the interval $[s, e]$ to a point $(\frac{s+e}{2}, e-s)$, i.e., to its center and length. While queries restricting the size of the objects are now handled by a range restriction on the dimensions the interval length was mapped to, all other important query types map to "parallelograms", i.e., not to axis parallel range queries (see Figure 7.4).

Unless an index supports this kind of restrictions, it is necessary to approximate them by a set of query boxes (Figure 7.5(a)). This usually decreases the performance in comparison to a single query box due to either many more query boxes or a bad approximation. It is also possible to restrict the queried area by taking the maximum object size into account (Figure 7.5(b)) and thus reduce the number of query boxes or enhance the approximation by smaller query boxes covering less space. The approximation can be further enhanced by taking the distribution of object extents/lengths into account. If there are more smaller objects the corresponding range queries should be thinner (Figure 7.5(c)).

Due to these reasons, the midpoint transformation is not suited well for mapping spatial objects when using the UB-Tree for indexing the parameter space. Rather than using the midpoint transformation, one should consider adding one or more additional dimensions for the object extents to the UB-Tree or use a separate secondary index.

7.2.1.3 Mid Point + Max Length Transformation

In order to reduce the number of dimensions for the mid-point transformation, one can only store the maximum length instead of all lengths. This reduces the number of required dimensions to $d + 1$ at the cost of a bigger candidate set. For data sets where the extent is similar in both dimensions, this will result in only slightly bigger candidate sets, while enhancing the clustering due to fewer dimensions.



(a) IQ approximation (b) Constricted IQ (c) Constricted approximation

Figure 7.5: Constricted Intersection Query and approximations

7.2.2 Approximation by SFC-segments

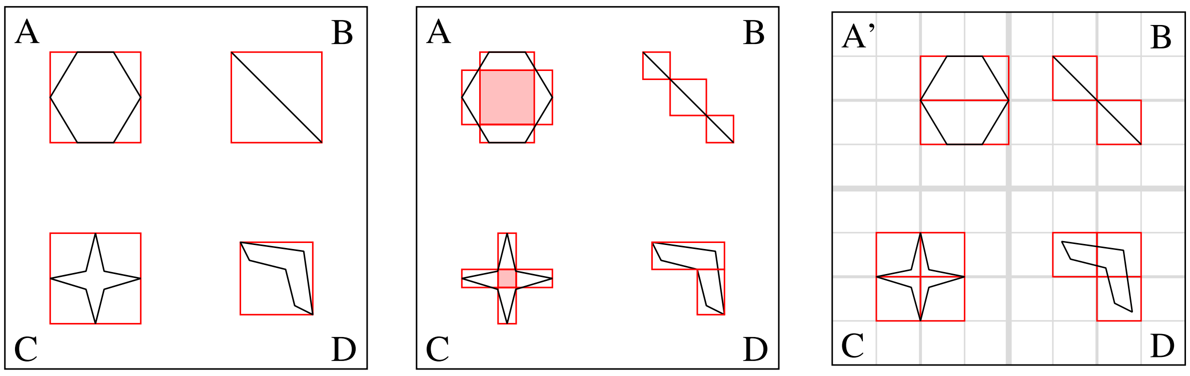
A single MBB cannot provide a good approximation for all objects shapes. To get a better approximation one can use a set of MBBs, but this requires to decompose the object which is not trivial as there are many different decompositions when choosing MBBs freely and even more when allowing overlap.

One solution to make the decomposition efficient is to utilize the quad-tree partitioning and use MBBs for the sub-cubes contained within or intersecting the object. The higher level sub-cubes completely contained within the object are selected as MBB and for sub-cubes which are only intersected the quad-tree partitioning, recursion is applied.

A partitioning down to the level of the finest granularity may result in many small MBBs, in case of a diagonal even a MBB for each point is created. This is costly w.r. to time required for calculating the approximation and storage space for the MBBs. Thus either the granularity or the number of MBBs should be restricted to limit the required resources. MBBs of the same size which are side by side can be merged into a single MBB thus reducing the number of MBBs, but this is not possible if they differ in size or are not side by side as for three MBBs arranged in a `L` shape.

Example 7.1 (Approximation variants)

Figure 7.6 shows three types of approximation: a single MBB, a set of MBBs, and a set of SFC-segments. The four example objects are a hexagon (A), a diagonal line (B), a star (C), and a boomerang (D). Only the hexagon is approximated reasonably well by a single MBB in Figure 7.6(a). Approximation can be enhanced by a set of MBBs for each object as in Figure 7.6(b). Overlap can be allowed as the MBBs are referring to the same object (see colored center of object A and C). Using a quad-tree for the partitioning is much easier. However, collapsing a set of consecutive sub-cubes into a single SFC-segment, highly depends on the utilized SFC and the location of the object, e.g., at its original location object A would decompose into four segments (like object C), but when shifted as in Figure 7.6(c) it requires only two segments.
◇



(a) MBB (b) Set of MBBs (c) Set of Z-Segments

Figure 7.6: Approximation Variants

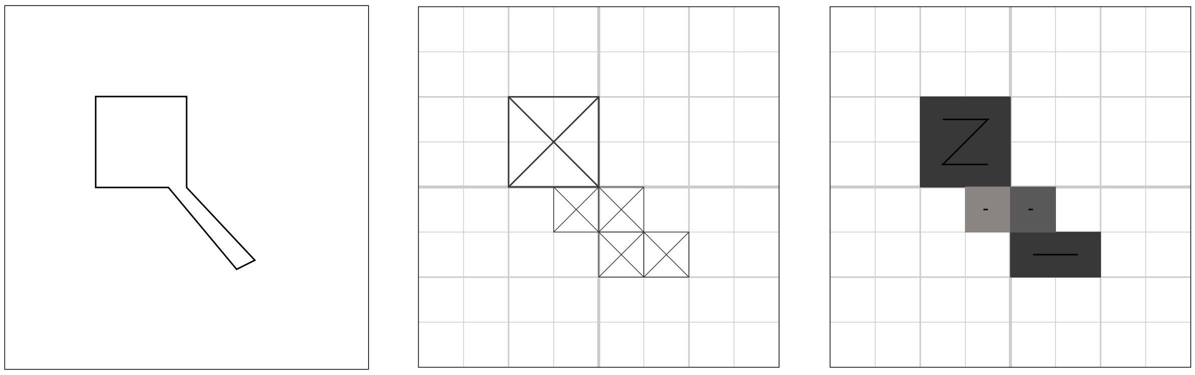

(a) Object (b) Quad-Tree (c) Z-Curve Segments

Figure 7.7: SFC-Approximation

The quad-tree already provides the basis for a SFC partitioning as in Figure 7.7(b). Thus, instead of MBBs one can also use it and the corresponding SFC-segments instead of MBBs. A sequence of adjacent SFC segments can be merged into a single segment and thus storage requirements can be reduced (Figure 7.7(c)). This further allows to use more segments in order to achieve a better approximation. The actual partitioning into quad-tree nodes is a breath first traversal down the tree pruning all path that do not intersect the object. Traversal stops if node is fully contained within the object. Further, a limit for the number of generated nodes should be applied in order to handle the tradeoff between storage requirements and quality of approximation. Taking an actual query workload into account, one can increase the number of SFC-segment for object which are false positives, i.e., candidates which are not part of the final result. Thus the number of false positives can be reduced. Another optimization is to take the neighboring objects into account during approximation, i.e., an approximation can be considered as good enough if it does not intersect with neighboring objects resp. their approximation anymore.

The resulting SFC-intervals can be indexed by a parameter space which in this case always has two dimensions, while the actual universe may have an arbitrary number of dimensions.

Queries are processed by decomposing the query shape into a set of SFC-intervals which will be checked for intersection (resp. the desired query type) with the SFC-segments in parameter space.

7.2.3 Indexing the Parameter Space

As previously mentioned the crucial question is, how well the UB-Tree can handle the highly skewed data distribution.

As the UB-Tree inherits the good properties of B-Trees it guarantees a 50% page utilization and usually it will be around 81% [Küs83, BY89]. Its Z-region partitioning adapts to the data distribution, i.e., densely populated areas are finer partitioned while the dead space above the main diagonal is covered by a few big Z-regions. Data structures like the GRID file would partition also the dead space, because of the fixed partitioning. Due to its disjoint Z-partitioning, point and range queries are always limited to only one search path in the B-Tree index, which is not true for the R-Tree due to the possible overlap of MBBs.

Figure 7.8(e) shows ten thousand intervals with uniformly distributed start and length. The resulting UB-Tree space partitioning is depicted in Figure 7.8(f). As one can see from the space partitioning, the UB-Tree adapts well to the non-uniform data distribution in parameter space. The Z-regions of the UB-Tree narrowly adapt to the data distribution without degenerating. The empty part of the parameter space is covered by a few huge Z-Regions, but those are not empty as they occupy some populated part of the universe with a small spot. The Z-regions cluster the intervals according to their position and length, which is reflected by their nearly rectangular shape. Therefore, it is likely that intervals within a Z-region have a similar position and length which is a benefit for query processing, since queries likely want to get these intervals together.

Also previous investigations [RMF$^+$01] have shown that the UB-Tree handles skewed data distributions well.

Additionally the UB-Tree is a dynamic index structure and supports updates with logarithmic performance guarantees.

For the end-point transformation all basic query types map to multidimensional range queries on the parameter space. The same holds for all the 13 Allen relations [AH85] for time intervals.

As the UB-Tree is designed to perform multidimensional range queries it is able to handle these query types efficiently. Furthermore, we need just one algorithm to deal with all these query type and there is no further need for specialized algorithms dealing with a specific query type.

The Z-region partitioning, which tries to maintain rectangular Z-regions with just a few fringes, fits perfectly to the query profile created by the interval queries. Therefore, the UB-Tree mainly loads those data pages from disk which contribute to the result.

7.3 Experiments: Interval Data

In order to compare the performance for one dimensional intervals we have used the end-point transformation with a parameter space indexed by the UB-Tree. As competitor we have selected the RI-Tree which has proven to be superior in comparison to other interval indexing techniques [KPS00, KMP$^+$01]. Further, both are designed to be implemented on top of an existing DBMS and to support secondary storage access.

For our measurements we use generated data and queries, as well as real world data.

We use a Sun Enterprise Server 450 with two Sparc-Ultra4 248 MHz CPUs and 512MB RAM. The secondary storage medium is an external 90 GB RAID system. The database system is TransBase HyperCube, which offers a product strength implementation of the UB-Tree.

7.3.1 Artificial Interval Data Sets

The data type of start and end is an integer with the domain $[0, 2^{20} - 1]$ and to each tuple we add an additional payload field of 200 bytes, resulting in a tuple size of 208 bytes. With a page size of 2kB this makes 9 tuples per page. The page size is small compared to modern DBMSs, but it allows to measure results w.r. to accessed pages qualitatively similar those obtained by measuring with larger data sets.

In order to get an insight view and understanding of the measured indexing techniques we use four different artificial data sets and two different artificial query sets.

In order to reflect different applications scenarios we use four data distributions. The start position is always uniformly distributed on the interval domain, while the length was varied. We used data set sizes of 1000, 10000, 100000 and 1 million tuples.

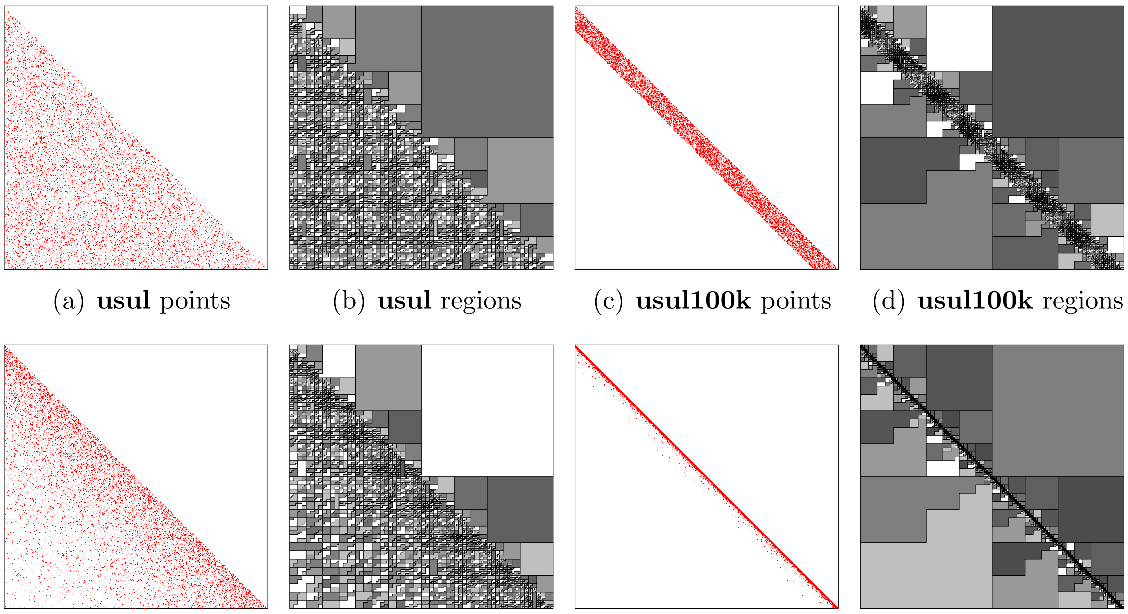
usul: uniformly distributed start and length

usul100k: uniformly distributed start and uniform distributed length within the range $[0, 100000]$

usel: uniformly distributed start and exponentially distributed length according to the exponential distribution function $\lambda e^{-\theta x}$ where $\lambda = 0.000476837$ and $\theta = 5.24288$

usel100k: uniformly distributed start and exponentially distributed length within the range $[0, 100000]$ according to the exponential distribution function $\lambda e^{-\theta x}$ where $\lambda = 0.002765655$ and $\theta = 367.0016$

The uniform distribution on start and end is used as it allows easier understanding of effects due to the availability of a cost model for UB-Trees. The exponential distribution of the length reflects most real world applications where short intervals are more likely to occur than long intervals. Furthermore, in real world applications there is usually an upper


(a) **usul** points (b) **usul** regions (c) **usul100k** points (d) **usul100k** regions

(e) **usel** points (f) **usel** regions (g) **usel100k** points (h) **usel100k** regions

Figure 7.8: Interval Distribution and corresponding UB-Tree Z-region partitioning

bound for the interval length[1] and therefore we use also variants of the data distributions that restrict the interval length to a given range.

7.3.2 Artificial Interval Query Sets

We focus on intersection queries, since exact match queries are trivial. The results for intersection queries also hold for the containment and enclosure query, as those are a subset of the intersection query.

We use two different query sets:

window scan: a set of 5000 queries sorted according to the start point and with a length of 300. This results in a query set covering the whole data space, while two consecutive query intervals have an overlap of ≈50%.

random: 1000 random query intervals, where the intervals have been taken from the **usel** data set.

The *window scan* query set is used to locate performance dependencies of the index structures which are based on the location of the query interval. On the other hand it gives the index structures/DBMS a chance to benefit from caching and clustering, due to the overlap of query intervals.

[1]When storing transaction time intervals, they do not last forever since transactions will be aborted after a timeout period.

The *random* query set is used to see how the index structures perform without caching. It shows how good the index structures handle ad-hoc queries, which may deteriorate caching.

7.3.3 Interval Index Structures

For our comparison we have used the composite key index (B-Tree) and UB-Tree provided by TransBase and variants of the RI-Tree [KPS00, KMP$^+$01] as a reference candidate for indexing techniques explicitly designed for intervals. Multiple secondary indexes are neglected as they do not provide clustering and their performance rapidly degenerates with bigger result sets due to random accesses on the base table, which can been observed in test measurements.

COMP: a composite key index on $(start, end)$

RIS: RI-Tree with two secondary indexes, one on $(node, start)$ and the other on $(node, end)$

RIP: RI-Tree with primary composite key index on $(node, start)$ and a secondary index on $(node, end)$

UBPS: UB-Tree on $start, end$

The RI-Tree was chosen, since it provides the same practically important properties as our approach: it is easy to implement/integrate, uses standard DBMS methods and provides scalability, update-ability, concurrency control, space efficiency, etc. Furthermore, it has been proven to be superior to the Window-List [Ram97], Tile Index (T-Index) [Ora00c, Ora00a] and IST-technique [GLO$^+$96] and so we can compare our performance results to these indexing techniques.

Actually, we first implemented the RI-Tree as it was presented in [KPS00] with two secondary indexes and “transient” tables, but the performance was worse than that of multiple secondary indexes. So in order to cluster the data we replaced the secondary index on $(node, start)$ by a primary index on $(node, start)$ [KPS01]. Still the performance was not as good as expected. Actually, the problem was that TransBase does not support transient tables and the join of *leftnodes* and *rightnodes* with the base table is not handled as efficient as our solution, which was to embed the $leftnodes$ and $rightnode$ list as a `IN` clause in the SQL statement. The maximum length of the list is $\log_2(max - min) - 1$ and for the data used in our measurements it was 19. However, the average length was just 8 and using an array search on such a short array is more CPU efficient than the operations caused by a join.

7.3.4 Qualitative Comparison of Interval-Index Structures

As we have seen before the UB-Tree handles all the discussed query types with its range query algorithm. This is superior compared to other techniques that support just a few or one query type, e.g., the RI-Tree as presented in [KPS00] handles just intersection queries. In order to handle other query types one has to use a combination of intersection query and post filtering or develop specific algorithms.

Furthermore, when indexing additional attributes one may just add them as additional dimensions to the UB-Tree. As UB-Tree clusters data symmetrically w.r. to all indexed attributes, range restrictions on them will also be handled efficiently. When using the RI-Tree one has to add the attributes to its composite key index or use secondary indexes. However, most implementation of composite key indexes cannot handle multidimensional range queries efficiently, i.e., they only utilize the range restriction on the first indexed attribute, and secondary indexes do not perform well, since they do not cluster the data.

7.3.5 Interval Measurement Results

When making performance measurements of index structures it is important to take all operations into account and not just the query performance. Theses are loading, space requirements, clustering, queries, updates, locking, archiving, etc.

We have concentrated on the following ones: Loading, space consumption, query performance and clustering, as updates, locking and archiving are handled well by the B-Tree underlying the tested indexing techniques.

7.3.5.1 Bulk Loading Time

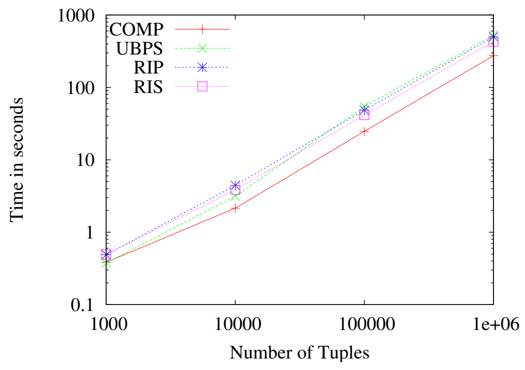
Figure 7.9(a) and Figure 7.9(b) are plots of the loading time. Loading time scales linear for all index structures. The data was sorted according to start before loading it, since this can usually be assumed for real world time data.

COMP loads fastest as it the core index structure of TransBase and has been highly tuned over the years. RIS follows as it only requires updates of its indexes while just appending data to the base table. The RIP index requires more time due to the clustering primary composite key index on $(node, start)$. UBPS is faster than RIP for small **usul** data sets and follows shortly after the RIP for all other data sets.

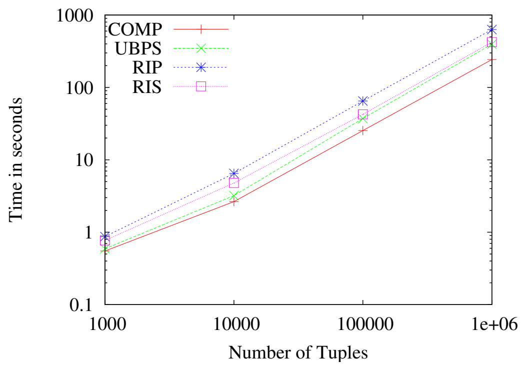
We have also performed bulk loading of unsorted data. The results of this were that UBPS was fastest, since it requires less sorting than COMP, which took 2.2 times longer in average. RIP and RIS followed with times longer by a factor of 2.32 and 2.35, due to their more expensive index maintainance and complex sorting.

7.3.5.2 Size of the Tables and Indexes

The overall size of the measured indexes is fairly the same and grows linearly with the data volume as shown in Table 7.1. The extra attribute *node* of the RI-Tree has only a minor influence, since four extra bytes do not contribute much to a tuple size of 208 bytes. However, it should be considered that tuples of the form $(start, end)$ would result in 50% higher space requirements, due to the extra *node* attribute.



(a) **usul** data set



(b) **usel100k** data set

Figure 7.9: Loading times with different scaling factors

count	COMP	UBPS	RIP = data + index end	RIS data + index start + index end
1000	148	150	135+11=146	150 + 11 + 10 = 171
10000	1450	1471	1315+87=1402	1471 + 87 + 76 = 1634
100000	14486	14681	13125+770=13895	14682 + 770 + 737 = 16189
1000000	143626	146788	138742+7345=146087	146795 + 7345 + 7349 = 161489

Table 7.1: Space Requirements

In general COMP and RIP require the fewest pages, followed by UB-Tree and RIS. UBPS requires slightly more pages compared to COMP and RIP, because compression of UB-Tree data pages has not been implemented in TransBase HyperCube at the time of the measurements. However, the differences can be neglected as they are minimal. The page utilization of the different techniques is usually between 77% and 90% and varies with the scaling factor and no best indexing technique can be recognized.

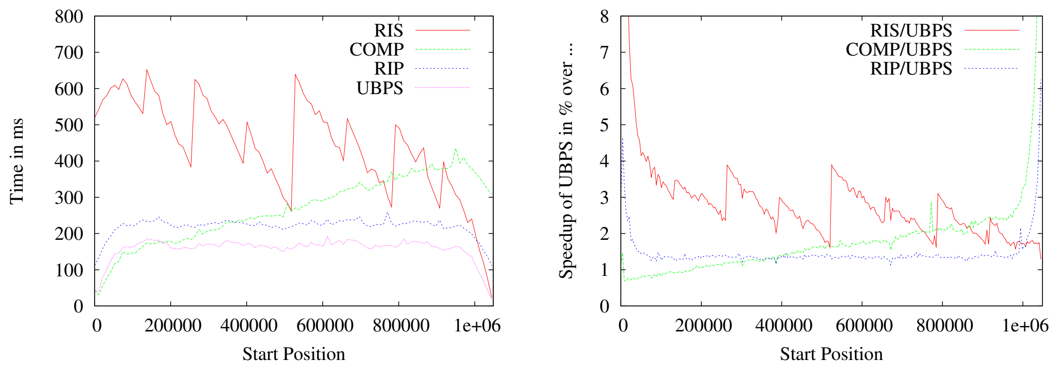
At the maximum scaling factor of 1 million tuples the DB size was around 290 MB in average and UBPS requires 2% more pages than COMP while RIP/RIS require around 1% more pages than COMP in average.

7.3.5.3 Query Performance

We use a data base size of 10000 tuples for the measurements. Test measurements with bigger data sets have shown the similar qualitative results as those presented here. First we want to consider the **window scan** query set.

Figure 7.10(a) shows the times for measuring the **usul100k** data set. All the measurements plots are sampled to fewer measurement points in order to easier distinguish the curves.

RIS is not a clustering index and we clearly see peaks which mark the nodes of the binary tree. When the scan window reaches a new RI-Tree node it causes random page accesses to disk pages containing tuples of that node. After that, these pages remain in



(a) Time (b) Ratio

Figure 7.10: Measurements with **homogen** query set on **usul100k** data set

cache and therefore the performance is better again until the window reaches the next node.

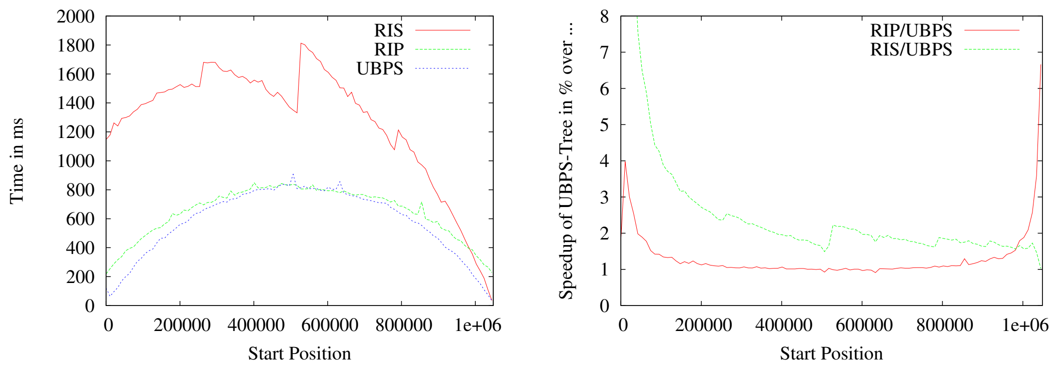
COMP reflects the fact that such an index is only able to exploit the range restriction on the first attribute *start*, but not the range restriction on the second attribute *end*. Due to the mapping of the intersection query, at the beginning of the domain it has a selective restriction on *start* and no restriction on *end*. As the query window shifts towards the end of the domain the restriction on *start* decreases while the restriction on *end* increases. Finally, only *end* will be restricted.

As we have seen in Figure 7.10(a), the times for COMP and RIS are not only related to the result set size but they also show a high relationship between the position of the query window and the response time. We refer to this as *positional dependency*. RIP and UBPS show a better performance and the response time reflects no positional dependencies, but it is linear to the result set sizes (plot omitted). As there are less intervals at the beginning and end of the interval domain they are faster there.

Figure 7.10(b) shows the ratios between COMP/UBPS, RIS/UBPS and RIP/UBPS. In average RIP requires 46% more time than UBPS and in the worst case RIP required nearly 8 times longer than UBPS. Especially for small result sets UBPS outperforms RIP.

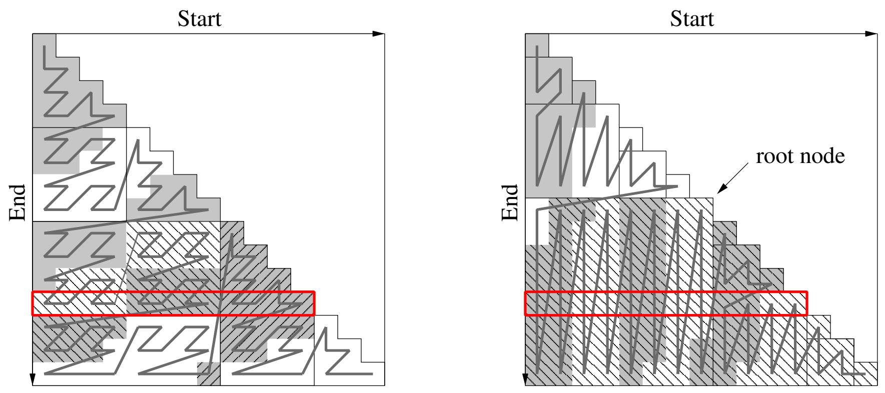
The measurements with the **usel100k** data set have similar results, due to this we omit them. In general the response times have been shorter as the result set sizes are smaller, because there are more shorter intervals in comparison to the data sets which do not restrict the length of the intervals. Due to this there are also more intervals assigned to lower nodes of the binary tree of RIS/RIP and RIS shows more peaks, but not high ones. This holds also for RIP, while UBPS maintains its stable behavior.

Figure 7.11(a) shows the measurement for RIP and UBPS with the **usel** data set. The results are similar as before, however the difference between RIP and UBPS becomes less. In this measurement RIP requires 30% more time in average and RIS was three times slower than UBPS. Again RIS shows the positional dependency. As before these results are also qualitatively observed for the **usul** data set.



(a) Time usel (b) Ratio usel

Figure 7.11: Measurements with **homogen** query set on **usel** data set



(a) UBPS (b) RIP

Figure 7.12: Clustering and Page Partitioning

As the performance of COMP depends on the position of the query interval we do not consider it further, but we focus on RIP and UBPS which perform linear to the result set size.

The **random** query set performs as follows. With small result set sizes UBPS is up to five times faster than RIP and 8 times faster than RIS. RIS shows again the unpredictable varying response times due to missing clustering. With growing result set size RIS and RIP become better and finally RIP is even 5% faster than UB-Tree, which comes from the better page utilization of the composite key index which is used for RIP. In average UBPS performs just 2% better than RIP.

7.3.6 Clustering

In order to get a better understanding on the differences between UBPS and RIP it is crucial to investigate their clustering and how it differs.

Figure 7.12 shows the region partitioning (and also disk page clustering) for the UB-Tree and the RI-Tree with a clustering composite index on (node,start,end)[2]. The domain for this picture is [0 : 14] and we assume a uniform distribution on *start* and length of the intervals. A *ends-within* query (all intervals that end at a given range) is also depicted and the pages which have to be loaded are filled with a stripe-pattern. In this case the RI-Tree has to load twice the data pages (6) of the UB-Tree (3).

The UB-Tree clusters the data symmetrically, i.e., it treats *start* and *end* of an interval equally. This results in clustering intervals w.r. to their position and length at the same time. The RI-Tree with a primary index on $(node, start, end)$ clusters according to *node* and then to the composite key order resulting in a stripe like partitioning. With growing data volume the UB-Tree would adapt its regions by making them smaller and more rectangular like. The RI-Tree will get even more strip like and a query like presented here will cut it like a puff-pastry, while the UB-Tree provides a good approximation for the query.

Therefore, it cannot efficiently support the *meets*, *left-overlaps*, *left-covers* relations [AH85] are not handled well by a RI-Tree with the primary index on $(node, start)$ resp. *met by*, *right-overlaps*, *left-covered* are not handled well by a RI-Tree with the primary index on $(node, end)$. The reason for this is that those queries restrict just the *start* resp. *end* cannot be processed well by a clustering index on the other (not restricted) attribute. Using the secondary index results in random page accesses and depending on the DBMS even multiple page accesses. Therefore, an index scan on the clustering index is usually more efficient, but it results in a complete scan of all affected nodes. Compared to this, the UB-Tree Z-region space partitioning allows for reading a good approximation of the subspace which contains intervals contributing to the result. With more data the UB-Tree will become even better compared to the RI-Tree.

However, a query restricting just start can be processed more efficiently by a RI-Tree on $(node, start)$, but also the UB-Tree can perform these queries and the RI-Tree is not able to outperform the UB-Tree by orders of magnitude.

7.3.6.1 Measurements

We use a **usul** data set with 100000 tuples for this measurement and move a restriction on *end* with length 10000 in steps of 10591 from the start of the domain to the end of it. Caching was enabled for this measurement. We also included the plots for multiple secondary indexes on a table without a specific sort order.

For the RI-Tree we used a modified version of the IQ query algorithm collecting just those nodes that might contribute tuples to the result, i.e., it works like the intersection query for intervals which are points, but is has to traverse only one path. This was done by traversing the virtual binary tree and collecting all those nodes with intervals which may contain the *start* resp. *end* point.

[2]Taking *end* not into account for the clustering of the RI-Tree would add random jumps to the space filling curve of the RI-Tree, since there would not be an order on *end*.

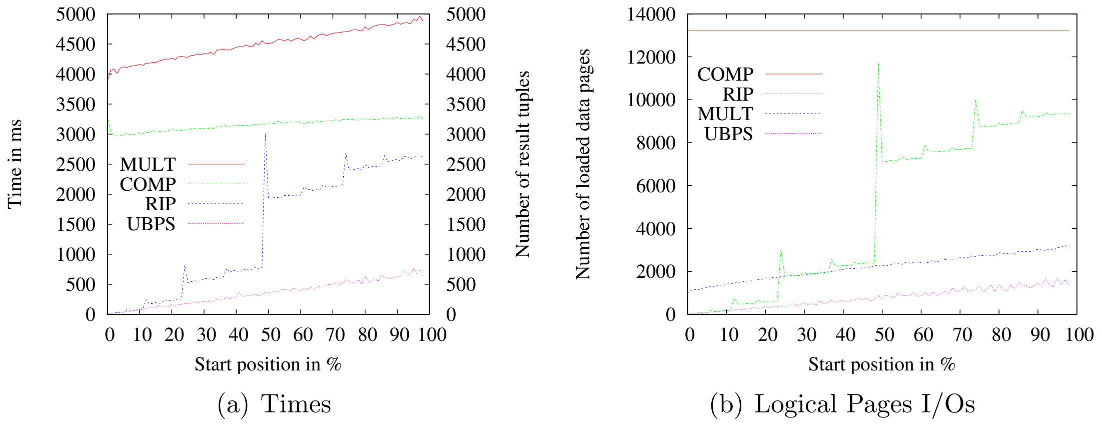
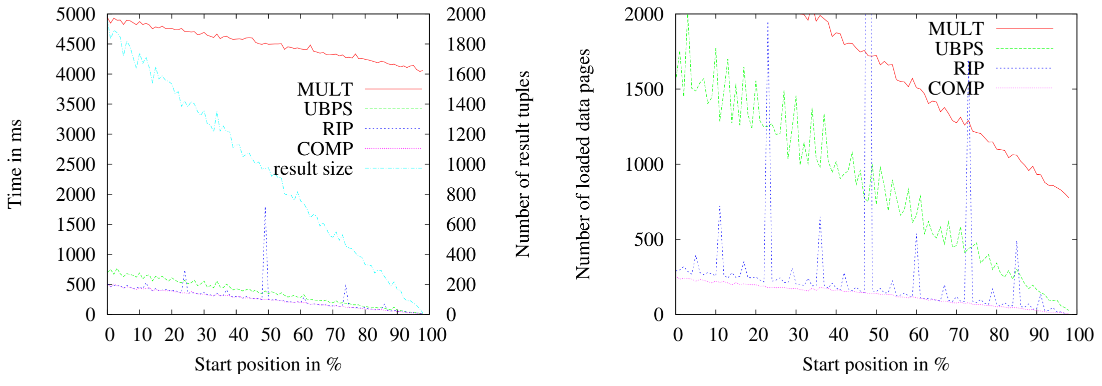


Figure 7.13: *Ends Within* query

Figure 7.13 shows the results for a query restricting only the end position. By shifting the restriction to the end of the domain the number of result tuples grows linearly, since there are more longer intervals. UBPS times reflects the linear dependency to the result set sizes. RIP starts similar to UBPS, but the further the restriction moves the more nodes containing more tuples have to be scanned for results. Again we see the phenomenon of peaks we have only seen for RIS before. Whenever, the restriction shifts over a new node the database has to fetch all affected pages. The first two levels of the binary tree are clearly visible at 50% on the first level (root node) and 25% and 75% on the second level. At 50% there is the highest peak, since suddenly the root node has to be processed, which was not cached so far. Additionally, also pages from the nodes before the root node are affected by this query. COMP on $(start, end)$ has to perform an index scan, since its implementation does not support a range restriction on end. Its slight increase comes from the growing amount of tuples which have to be transfered to the application. Figure 7.13(a) depicts the linear relation between response times and number of loaded pages. MULT performs really bad. It loads fewer pages than COMP, but it has to fetch them by random accesses and therefore it is not able to exploit prefetching as COMP does, further more it has to load more pages than UBPS or RIP due to the lack of clustering. Speaking in averages, MULT is nearly 13 time slower than UBPS, COMP is 9 times slower than UBPS, where RIP is 4 times slower in sum.

The results for a query restricting just *start* in the same way as *end* was restricted before are depicted in Figure 7.14. As expected COMP performs best, since it is perfect for this query type. It performs linear to the result set size. RIP is only a bit slower, but again with peaks at the places where the query window covers new and more nodes. UBPS follows again performing linear to the result set size. MULT is again not suitable for the same reasons as before. Speaking in averages, MULT is nearly 12 time slower than UBPS, while COMP takes 67% and RIP 75% of the time UBPS required.

To summarize: RIP requires 400% of UBPSs time for *ends-in* queries and 75% for *starts-in* queries. Therefore, UBPS is the overall winner.



(a) Times (b) Logical Pages I/Os

Figure 7.14: *Starts Within* query

7.3.7 Real World Data

For measuring the performance with real world data we use the log files of our WEB server. We took the time when a request came into the system as *start* and the time when the data transfer was completed as *end*.

There are 1.4 million tuples in the data set and each tuple has the following attributes:

Attribute:	start	end	hostid	fileid	filesize	logline
Datatype:	integer	integer	integer	integer	integer	char(200)

The attributes *fileid* and *hostid* had been normalized and hierarchically clustered with MHC [Pie03] according to the hierarchy of the dimensions, i.e., the path of the file and the reverse host name.

Figure 7.15 shows the data distribution of the intervals in the accesslog file. The time granularity was seconds and the time range 1997/1/1 to 2005/6/1 with data from 1997/12/1 to 2001/10/15. There were 23083 different files and 14441 different hosts.

As indexes we have used a four dimensional UBPS and RIP on (start, end, fileid, hostid) while parameter space resp. RI-Tree algorithms were only applied to the attributes *start* and *end*. The size of the bulk loaded indexes was 203351 pages for UBPS and 221553 pages for RIP, which is $\approx 9\%$ more due to the key of RIP which requires more storage.

As query suite we have used three different query sets which only differed in their time restriction. The other two dimensions are always restricted as follows: *hostid* was restricted to host from the toplevel domain `EDU`, which were 2489 different hosts and *fileid* was restricted to the files in the directory `/results`, containing papers, visualizations etc., 442 files all together. Time was restricted as follows:

homogen-hour: a time window restricted to one hour and shifted from min to max.

homogen-day: a time window restricted to one day and shifted from min to max.

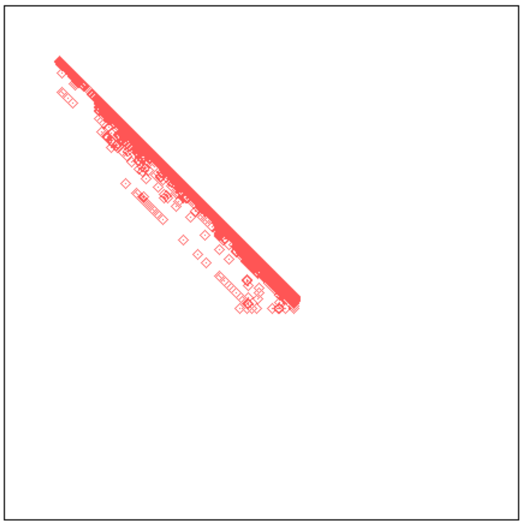


Figure 7.15: The data distribution for the Access Log

random: 1000 random queries.

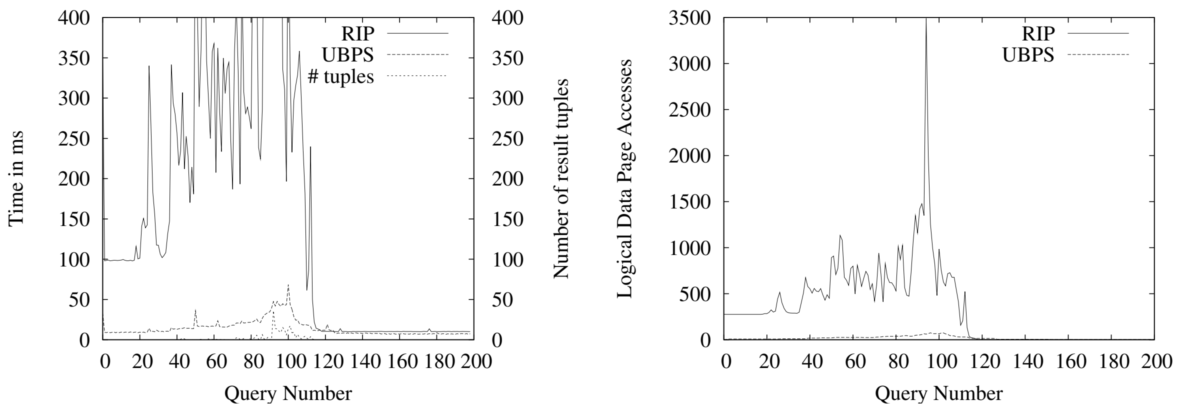
All measurements have been performed on top of TransBase HyperCube with clean caches before starting a measurements.

Figure 7.16, Figure 7.17, and Figure 7.18 show the results. We have limited the y-axis in the plot to 400ms in order to still see the UBPS. The maximum times of RIPS were 1845 for Figure 7.16(a), 3464 for Figure 7.17(a), and 1100 for Figure 7.18(a). After query 118 there were only empty results for the *homogen-hour* resp. *homogen-day* restriction as there was no further data.

UBPS outperforms RIP by several orders of magnitude due to its symmetrical clustering w.r. to all dimensions and its ability to also utilize the restriction on *hostid* and *fileid*. UBPS also shows a correlation between the time required for a query and the number of result tuples resp. loaded data pages. For RIP there is no significant correlation, which is best seen in the plot for the random query set (Figure 7.18).

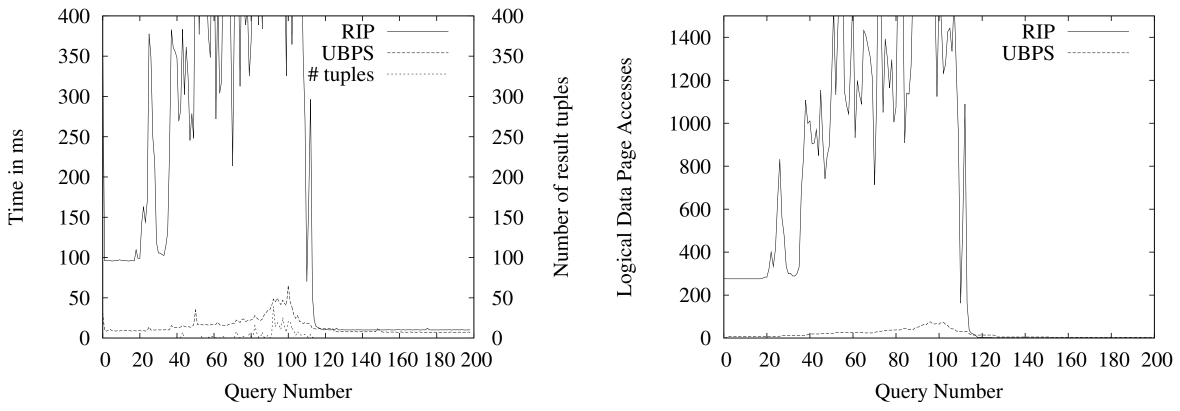
RIP performs successive index scans and thus only causes index page accesses for resetting the cursor. While UBPS loads twice the number of logical index pages than RIP, the index is smaller and those index pages are usually cached leading to fewer resp. equal physical index page accesses in total.

With increasing result sets for a whole day (Figure 7.17), RIP requires even more time for post-filtering result tuples w.r. to the other restricted dimensions. Only for empty results (after query 118) RIP shows a performance similar to UBPS.



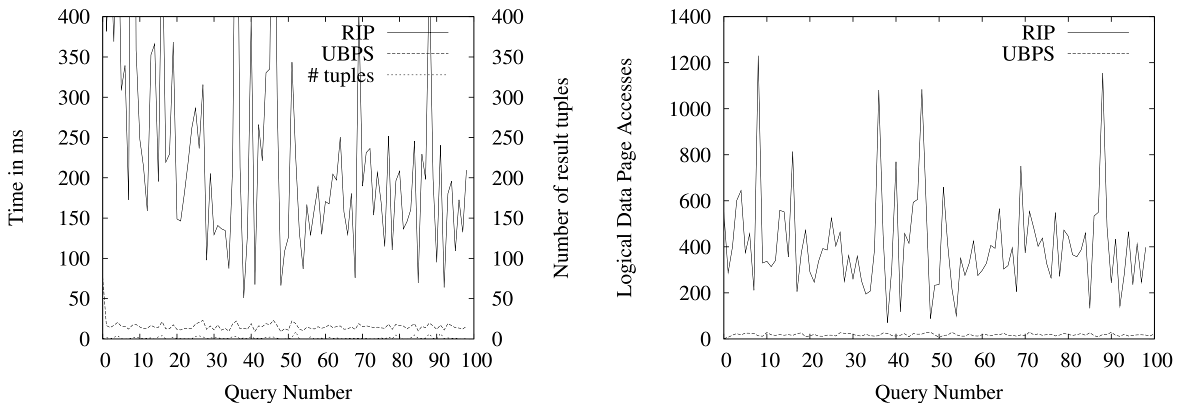
(a) Time (b) Data Pages

Figure 7.16: Homogeneous query window on hour



(a) Time (b) Data Pages

Figure 7.17: Homogeneous query window on day



(a) Time (b) Data Pages

Figure 7.18: Random Queries

7.3.8 GEO Data

The data set consists of polygons of objects in a city in Poland, e.g., areas, buildings, power lines, and streets. The indexed data are the MBBs of the buildings. While the R*-Tree is two-dimensional, the UB-Tree and BUB-Tree are four-dimensional for end-point and mid-point parameter spaces. This is an important difference in comparison to point data where the compared indexes always have the same number of dimensions as the indexed universe. A direct consequence of this is a reduced index page capacity and thus a reduced fanout for the UB-Tree and BUB-Tree.

Further, in comparison to point data the overlap of MBBs in the R*-Tree increases as we are now indexing MBBs of spatial objects which cause significant overlap and are more complicated to partition during a page split.

The 144155 MBBs of buildings are relatively small in comparison to the whole map and most of them are also more or less aligned to the axis due to the main streets which are running either from north to south or east to west. This lead to a good approximation of MBBs for buildings. Power lines, streets and plot areas are not so small and well aligned causing MBBs that often do not approximate them well.

More details on the properties of the data and plots of the original polygons and the resulting MBBs can be found in Appendix B.2.

In this section we summarize results from experiments performed together with supervised students as well as present measurements conducted by the author.

7.3.9 Endpoint Transformation

We have started with experiments on Oracle 8.1.3 comparing the spatial cartridge with a UB-Tree implemented as UBAPI, i.e., on top of Oracle [Nic01]. Loading the UB-Tree was three times faster than the R*-Tree and Quad-Tree. Containment queries were up to twenty times faster with the UB-Tree. The R*-Tree cannot prune significantly more search paths for containment queries than required to inspect for intersection queries, as each MBB intersecting the containment query might contain relevant MBBs resp. objects. In contrast to this, the end-point transformation usually maps a containment query to a range query significantly smaller than for intersection queries.

The performance of intersection queries with the UB-Tree was by 20-70% better than both the quad-tree and R*-Tree, depending on the actual query work load and data set. These results did not fit to our theoretical expectations, i.e., the R*-Tree should not access significantly more pages than the UB-Tree for intersection queries. The explanation for this lies in the mapping of the R*-Tree to a relation in Oracle. This deteriorates the clustering and mapping R*-Tree traversal back to relations further decreases the performance.

Due to this we have also performed the same experiments on RFDBMS. Here the R*-Tree performs much better. With the standard query mapping the R*-Tree even slightly outperforms the UB-Tree, but using the constricted intersection queries, the UB-Tree and BUB-Tree perform equally to the R*-Tree.

We have also performed some measurements on the effect of the page size on the index performance. These measurements have been performed on the system described in Appendix A.3. As data sets we have used three variants:

- only the MBBs of buildings
- all MBBs of real objects: buildings, streets, power lines
- all objects, including the point objects

As queries we have used:

Sliding Window: A sliding window moving in a line-wise scan over the whole map.

Streets: The MBBs of the streets. This corresponds to queries requesting all objects along a street.

Power-Lines: The MBBs of the power line objects. This corresponds to queries requesting all objects along a street.

All databases have been bulk loaded with pages sizes from 256 bytes to 4KB = 4096B. For the BUB-Tree resulting in a page utilization of $\approx 95\%$. The R*-Tree and UB-Tree have been loaded with 100% page utilization. The R*-Tree was again loaded in Hilbert-order (*rs-h*) as well as Z-order (*rs-z*). The measured queries have been intersection queries. For the UB-Tree and BUB-Tree the constricted end-point transformation has been used.

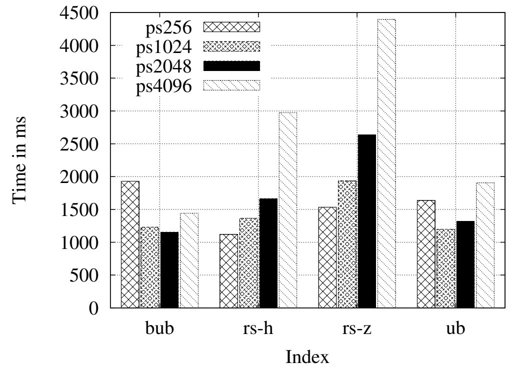
The results for the sliding window query without caching are depicted in Figure 7.19 and Figure 7.20 with enabled caching. For a page size of 256 byte, the R*-Tree variants outperform the UB-Tree and BUB-Tree w.r. to time and page accesses, causing only about half the data page accesses (Figure 7.19(b)). However, the advantage of the fewer page accesses is not directly reflected in the execution times. Further execution times increase for the R*-Tree variants with increasing page size due to two reasons: more overlap due to larger MBBs and higher cost for index page processing.

The UB-Tree and BUB-Tree benefit here by utilizing binary search on the index pages, while the R*-Tree always requires to inspect all tuples on an index page. For a page size of 4KB *rs-z* and *ub* have a similar number of data page accesses and the UB-Tree actually has slightly more index page accesses, but still it outperforms the R*-Tree by a factor of two w.r. to time. The BUB-Tree causes $\approx 30\%$ fewer pages accesses for 4KB pages than the UB-Tree and is also faster.

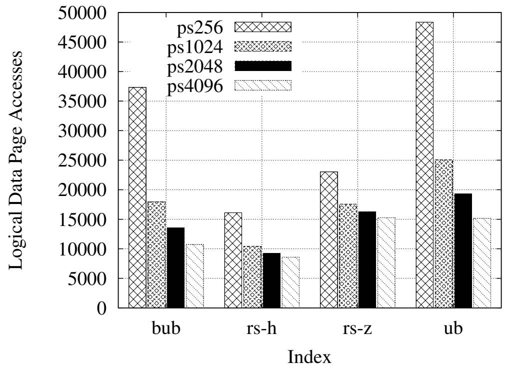
This is also reflected by the results of the same queries with enabled caching. While physical and logical page accesses have been essentially identical when disabling caching, the physical page accesses are now much lower (Figure 7.20(b)).

The differences change when querying a database containing all real objects[3] resp. all objects Figure 7.21(b). The range queries resulting for the end-point transformation become bigger now as the maximum extent of objects is larger now, i.e., street and power

[3]When speaking of *real objects*, we mean objects that correspond to an actual physical object, e.g., power lines, buildings, streets, etc. Other objects are not real but they are only used for graphical representation, e.g., captions, plot area.

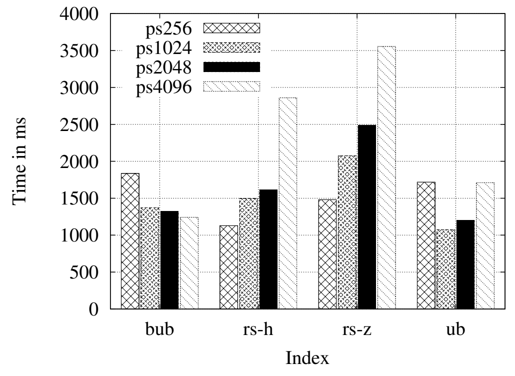


(a) (Building) time

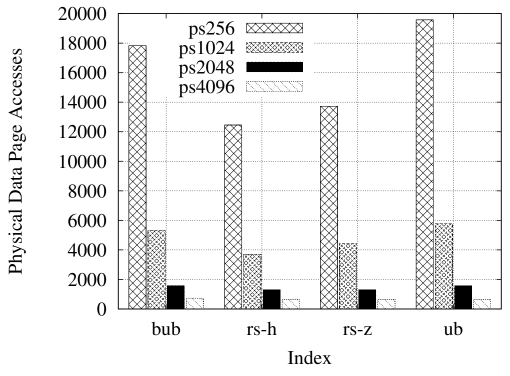


(b) (Building) lDR

Figure 7.19: Sliding Window on Building Data Set (without caching)

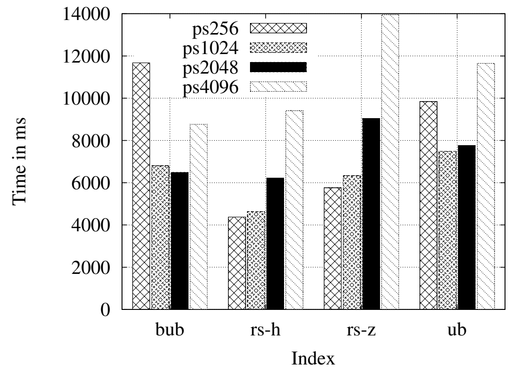


(a) (Building) time

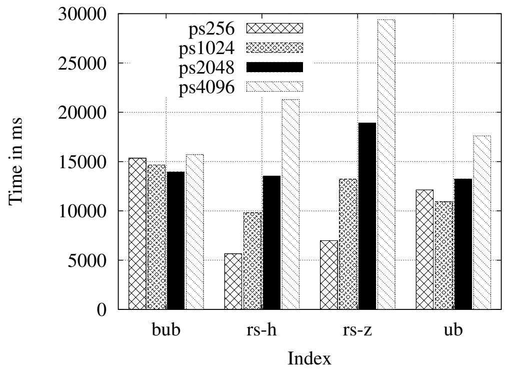


(b) (Building) pDR

Figure 7.20: Sliding Window on Building Data Set (with caching)



(a) (All Real Objects; Query = Streets) time



(b) (All Objects; Query = Power-Lines) time

Figure 7.21: Street-Queries and Power-Line-Queries (caching enabled)

line segments are longer than buildings. This reduces the performance for the BUB-Tree and UB-Tree for smaller pages sizes, but with increasing page size they again outperform the R*-Tree.

Including also all point objects and the plot-area objects reduces the differences again for small pages.

With even larger page sizes, 8KB up to 64KB pages as they are used in todays systems, the R*-Tree will perform even worse and page processing can become the bottleneck instead of I/O. On the other hand, the BUB-Tree and UB-Tree usually benefit from increased page sizes.

Further, for randomly loaded databases the R*-Tree does not show the advantage for small page sizes, due to more overlap. Again, one should note, that the R*-Tree is not suited for dynamic applications and due to our previous results on random loading (Section 6.13.5.2) we omit those measurements.

7.3.10 Comparison of Point-Transformation Variants

In [Unt02] we have investigated the transformations: end-point, mid-point, and mid-point with only the maximum extent. The resulting query shapes for the mid-point bases parameter spaces have been approximated by a set of query boxes which are processed by an optimized algorithm for UB-Trees developed by the author [FMB99]. All measurements have been performed with RFDBMS.

Only the building data set has been used for these measurements and taking only the maximum extent here works quite well as buildings usually have similar extents for both dimensions. The results show that using the constricted ranges results in a $\approx 20\%$ performance gain w.r. to time and accessed pages for the end-point transformation and up to 50% for the mid-point transformations. Further, taking the actual data distribution into account and thus use a finer partitioning in densely populated areas for the mid-point transformation variants directly pays off with significantly fewer page accesses (up to 50%). Indexing only the maximum extents and the mid-point results in $\approx 4\%$ increased candidate set while causing up to 20% fewer page accesses. Whether these savings in page accesses pay off depends on the cost of the final spatial filter. Still the mid-point transformations requires more page accesses and processing time for intersection queries than the end-point transformation or R*-Trees.

7.3.11 Approximation by SFC-segments

As noted before, there is a tradeoff between better approximation and higher space requirements due to a set of SFC-segments for each object.

In order to minimize the number of indexed SFC-segments for overlapping objects, two separate tables should be used. One table only storing the SFC-segments and indexed by a UB-Tree on parameter space and another table mapping SFC-segments to object IDs, with a compound B-Tree on the SFC-segments. Duplicated segments are then only stored once in parameter space.

If there is no object overlap, it is also possible to simply store both the SFC-segments and object IDs in the same table, but again with a UB-Tree only on the SFC-segments.

The results of our measurements shown in [Hod03] are as follows: The SFC-segments approximation is definitely better than a full table scan, which was also a result of [Gae95], but who did not present any comparison with an MBB based approach. Comparing the segments approach with the R*-Tree and the end-point transformation showed that there is no configuration were it can outperform the MBB approaches. Either the approximation is not sufficient causing more false positives, i.e., larger candidate sets. On the other hand, with an approximation better than MBBs the storage requirements increase drastically. While causing fewer candidates than a MBB approach, this causes more page accesses and higher processing cost due to more complex query statements, i.e., each query decomposes into a set of range queries.

We believe that the SFC-segment approach cannot provide better performance for the majority of real world data sets. It might only work with highly skewed object shapes, but in this case a decomposition into a set of MBBs will also offer a better approximation while being more flexible in the bounding, as MBBs do not require to be aligned to a quad-tree partitioning.

7.4 Summary

In this chapter we have discussed indexing techniques for spatial objects that utilize an existing UB-Tree resp. BUB-Tree by transforming approximations of the spatial objects and related queries to points.

Intersection queries are the basis for most other spatial queries. Theses are well supported by R*-Trees, thus a new spatial index has to compete against them. As alternative we have identified the end-point transformation in combination with a UB-Tree resp. BUB-Tree indexing the resulting parameter space.

For reasonable page sizes, the UB-Tree and BUB-Tree provide better performance for bulk loaded 2d real world data than R*-Trees while maintaining the properties of B-Trees: concurrency control, logarithmic worst case guarantees for the basic operations of insertion, deletion, and update. Although, having doubled the number of dimensions in comparison to the original space and a highly skewed data distribution, this approach still works well in combination with the UB-Tree and BUB-Tree.

This enables dynamic applications which are not possible with R-Trees, as their performance degenerate with random insertions due to increasing overlap. While R*-Trees may prevent the degeneration to some extent, they dramatically increase the cost of insertions. The UB-Tree and BUB-Tree do not suffer from such a design flaw as they guarantee an overlap free partitioning of the data. Therefore, point search is always bound by the tree height and guarantees the logarithmic cost of the basic operations.

In this context, the UB-Tree and BUB-Tree again show their universality. Not only point data, but also spatial objects can be managed and queried efficiently while allowing dynamic applications.

Chapter 8

Summary

In our research work presented in this thesis, we address the complex problem of multidimensional indexing. The UB-Tree claims to be universal by its name. We have taken a closer look at design decision of the UB-Tree and advanced applications. Before we come to our conclusion, we briefly summarize our contributions. All of our findings are backed by analysis and experiments with real-world data sets.

In Chapter 3, we have discussed the decision of selecting the Z-curve and underlying clustering technique. The Hilbert-curve provides better clustering on the level of tuples, but usually it is wrong to focus only on a single measure as this does not built a complete index. Page based access reduces its advantages. Addresses for universes are longer, when dimension cardinalities differ and thus cause a reduced index fanout. The address calculation requires significantly longer and calculating the intersection of a query box with the curve is exponential w.r. to the number of dimensions, while for the Z-curve its only linear to the address length. Therefore, when selecting a SFC for an application it is important to take all aspects into account, i.e., usually not only clustering.

In Chapter 4, advanced algorithms for bulk loading and deletion are presented. These are typical for DWH applications which are moving to real time DWH, i.e., while updating a cube monthly was sufficient, nowadays we want to be up to date on a daily basis. At the same time the amount of data which is collected increases tremendously and so also the databases grow rapidly. Our algorithms support this for the UB-Tree by efficient incremental operations.

In Chapter 5 the effects of clustering data indexed by a bitmap index is discussed. We point out the positive effects of clustering w.r. to index size and creation time as well as query performance. Without modifying the database schemata or application this allows for significant performance improvements by simply sorting the data according to a SFC order before loading it. Still the UB-Tree is superior to them.

In Chapter 6 we introduce a novel variant of the UB-Tree: the *bounding UB-Tree* . It is designed to handle dead space by storing information about it in the index while preserving the worst case guarantees of the UB-Tree. Dead space (unpopulated areas) occurs in nearly all real world datasets due to their skewed data distribution. We present algorithms for building, maintaining, reorganizing and querying BUB-Trees. In experiments with artificial and real world data we show its advantages compared to the UB-Tree. R-Tree variants can also provide similar performance, but they are not suitable for dynamic applications due to degenerating with insertions (increasing MBB overlap) and high insertion cost (R*-Tree) and without providing reasonable worst case guarantees. In the worst case, the whole database has to be read for an point query on a R-Tree. In contrast to this, the BUB-Tree has to traverse only one path in the tree.

In Chapter 7 we focus on indexing of spatial objects. The UB-Tree is only designed for point data. However, transforming spatial data and queries to points allows to utilize the UB-Tree for the indexing of spatial objects. Its query performance is comparable (sometimes even better) to index structures designed for spatial objects, e.g., the RI-tree for intervals and the R*-Tree for spatial objects. The major advantages of the UB-Tree resp. BUB-Tree are again their worst case guarantees for the basic operations and thus the ability to efficiently enable dynamic applications.

8.1 Conclusion

The UB-Tree is a flexible index structure providing excellent query performance while preserving the good properties of B-Trees. Integrating the UB-Tree into a DBMS kernel providing a B-Tree is much easier than integrating one of its multi-dimensional competitors and concurrency control and B-Tree optimizations basically come for free. The UB-Tree allows a diversity of applications and data to be efficiently supported.

Thus it is the true multi-dimensional successor of the B-Tree.

Appendix A

Description of the Test System

All workstations used for measurements are exclusively reserved in order to avoid any side effects. Herewith, manipulation of measured times due to CPU load and I/O from other processes is avoided.

A.1 System Sunbayer69

Type name	Sun Ultra 5
Manufacturer	Sun Microsystems
Number of processors	1
Main memory	512 MB
OS Version	SunOS Release 5.8
Hard Type	IBM-DNES-318350W-SA30
Hard Capacity	18 GB
Interface to peripheries	20 MB/sec (fast/wide) SCSI-2

A.2 System Sunwibas0

Type name	SPARC-Enterprise 450
Manufacturer	Sun Microsystems
Number of processors	2
Main memory	1GB
Operating system	SunOS Release 5.5.1
Hard Type	RAID
Hard Capacity	90 GB
Interface to peripheries	20 MB/sec (fast/wide) SCSI-2

A.3 System Atmistral7

Type name	AMD Mobile Athlon TM XP 2600+ 45
Manufacturer	Gericom
Number of processors	1
Main memory	512MB
Operating system	Linux 2.6
Hard Type	Toshiba MK8025GAS
Hard Capacity	80 GB
Interface to peripheries	ATA BM-DMA

Appendix B

Real World Data Sets

B.1 GfK Datawarehouse

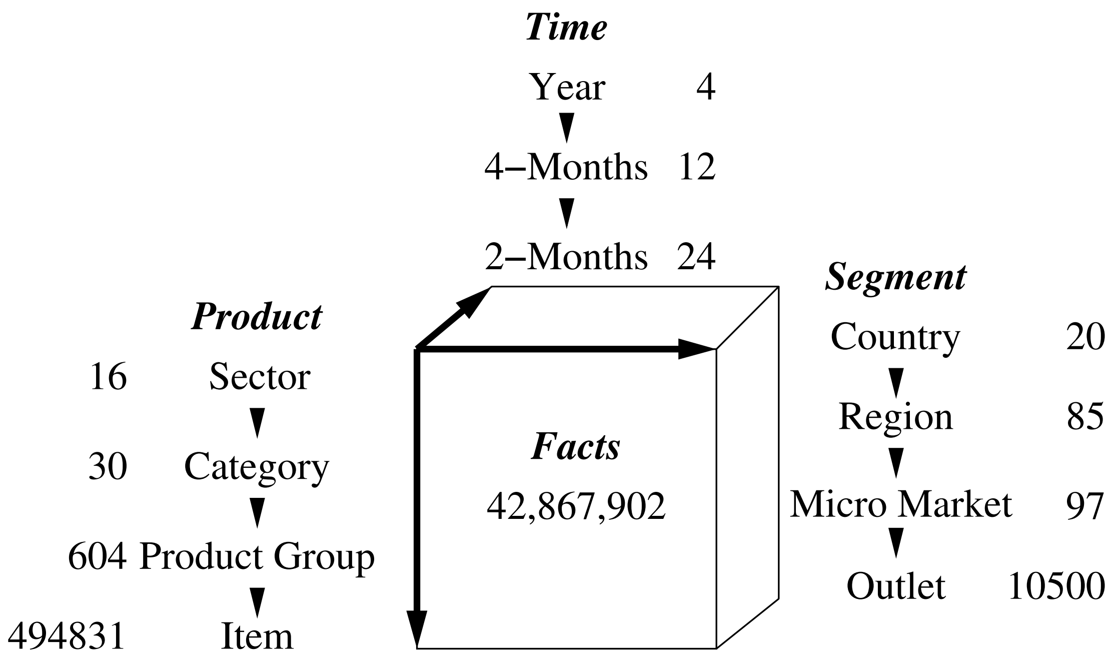
The Gfk Data Warehouse as modelled in the MISTRAL project has the three dimensions *Product*, *Segment* and *Time*. The hierarchies and the cardinalities of the dimension are depicted in Figure B.1(a).

The cardinalities of the dimension table items actually associated with fact table records is shown in Figure B.1(b). For the *Product* dimension it is significantly smaller than the cardinality of the dimension domain.

The data distribution for certain levels of the dimension-hierarchies can be found in Figure B.2, Figure B.3, and Figure B.4.

The resulting clustering, i.e., the location and order of tuples according to the Z-order is depicted for two dimensions by projection in Figure B.5, Figure B.6, and Figure B.7. Points are denoted by an + and connected by lines. Finally, Figure B.8 shows a 3d plot where the position of tuples w.r. to to Z-order is denoted by an gradient from black to white.

Further details on the data can be found in [Ram02].

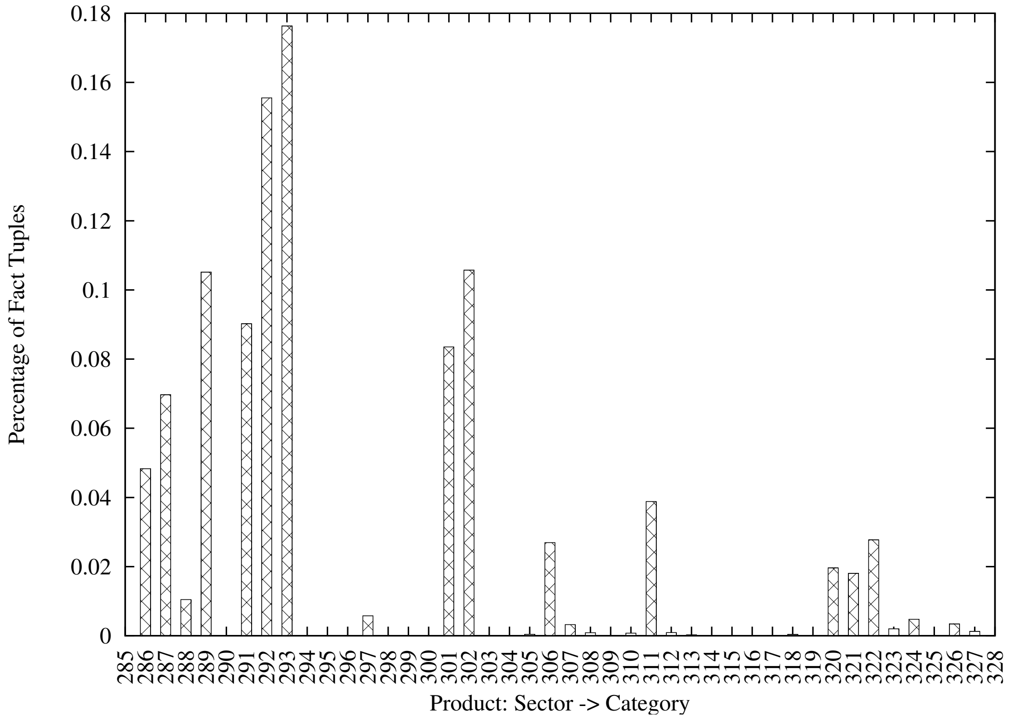


(a) Gfk Cube and Cardinalities of Dimension Hierarchies

Dimension	**Cardinality**
Product	360748
Segment	9556
Time	15

(b) Dimension Table Items associated with fact Tuples

Figure B.1: GfK Data Warehouse Cube and Cardinalities



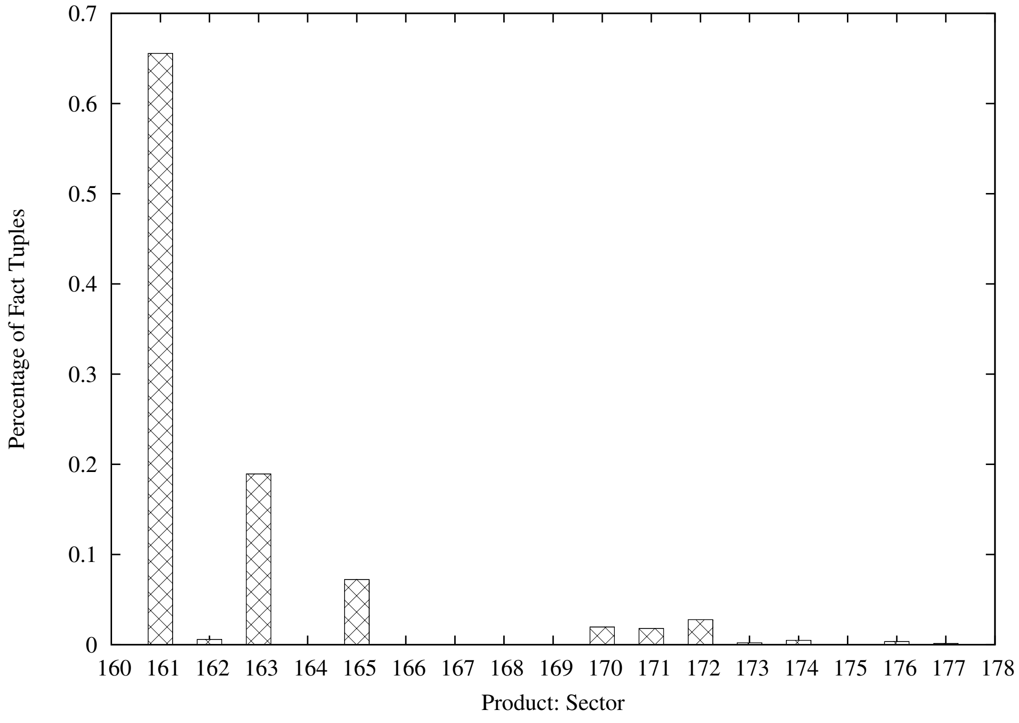


Figure B.2: GfK Data Distribution: Product

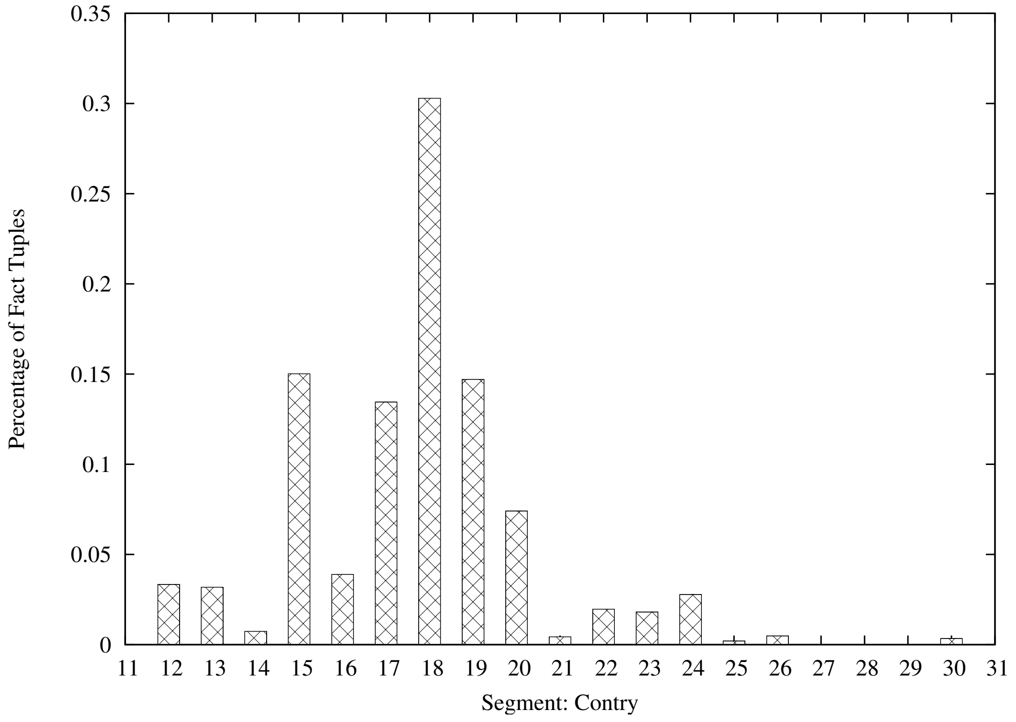
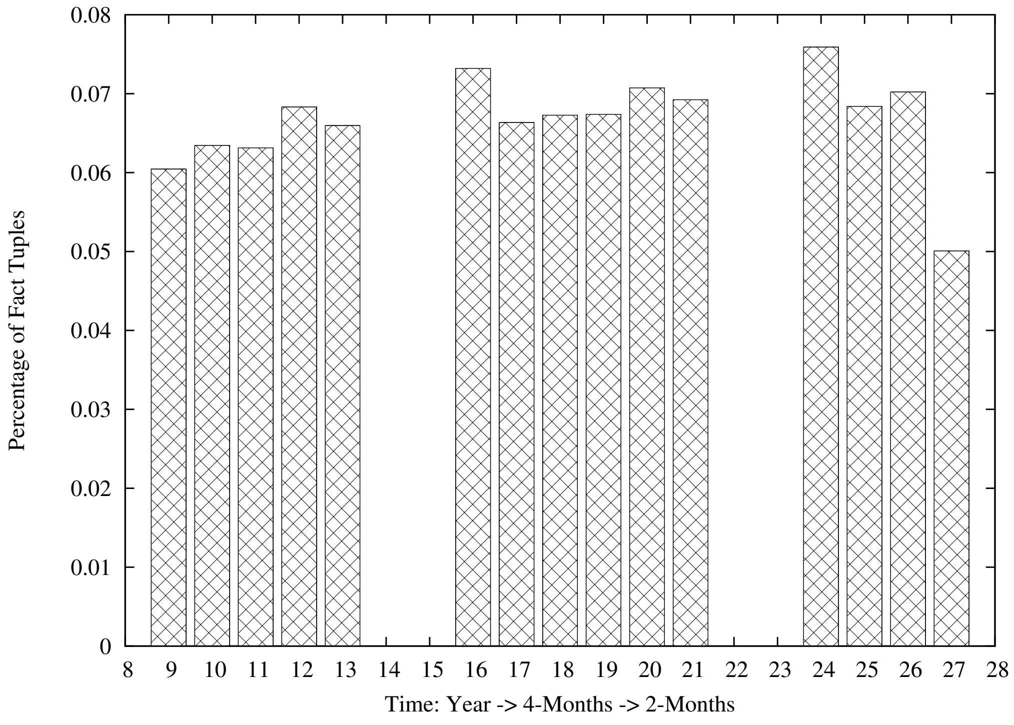


Figure B.3: GfK Data Distribution: Segment



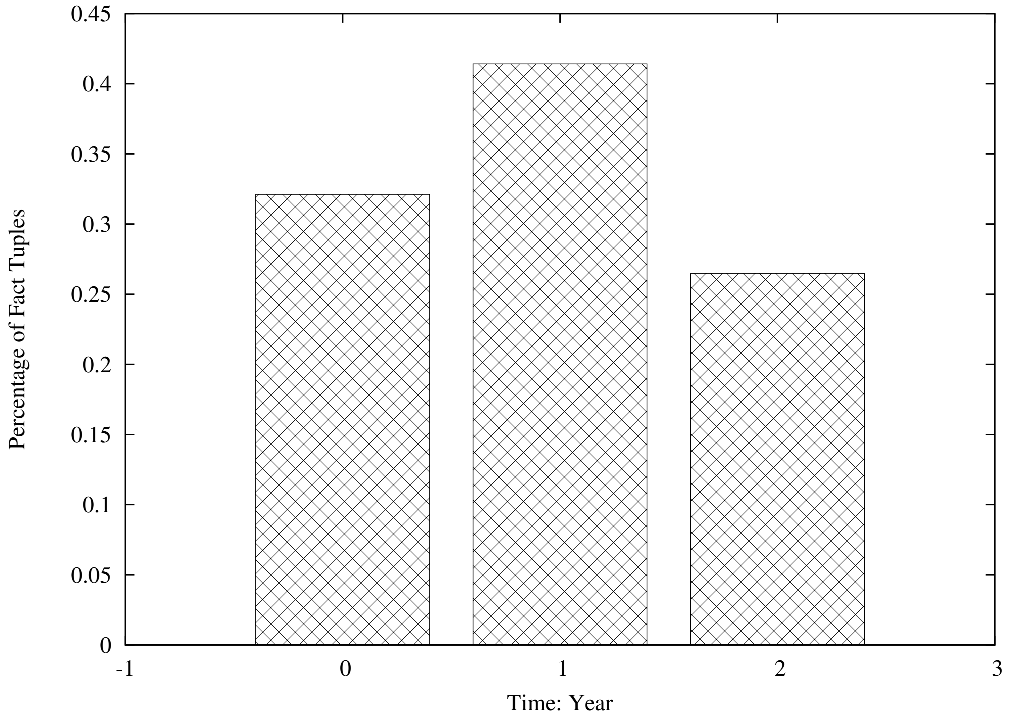


Figure B.4: GfK Data Distribution: Time

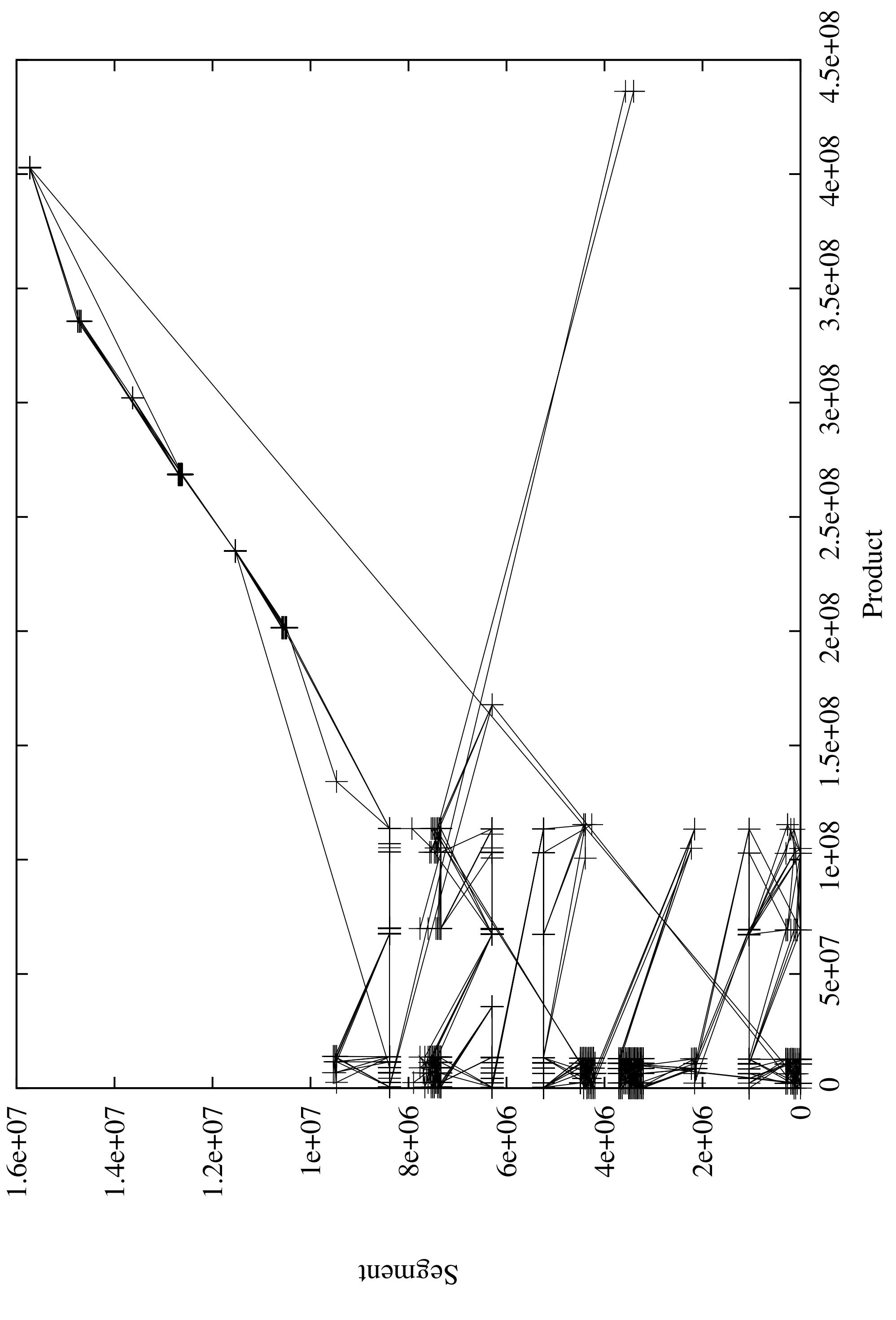


Figure B.5: GfK Data Distribution: 2d Projection of (Product,Segment) with points connected by a line in Z-order.

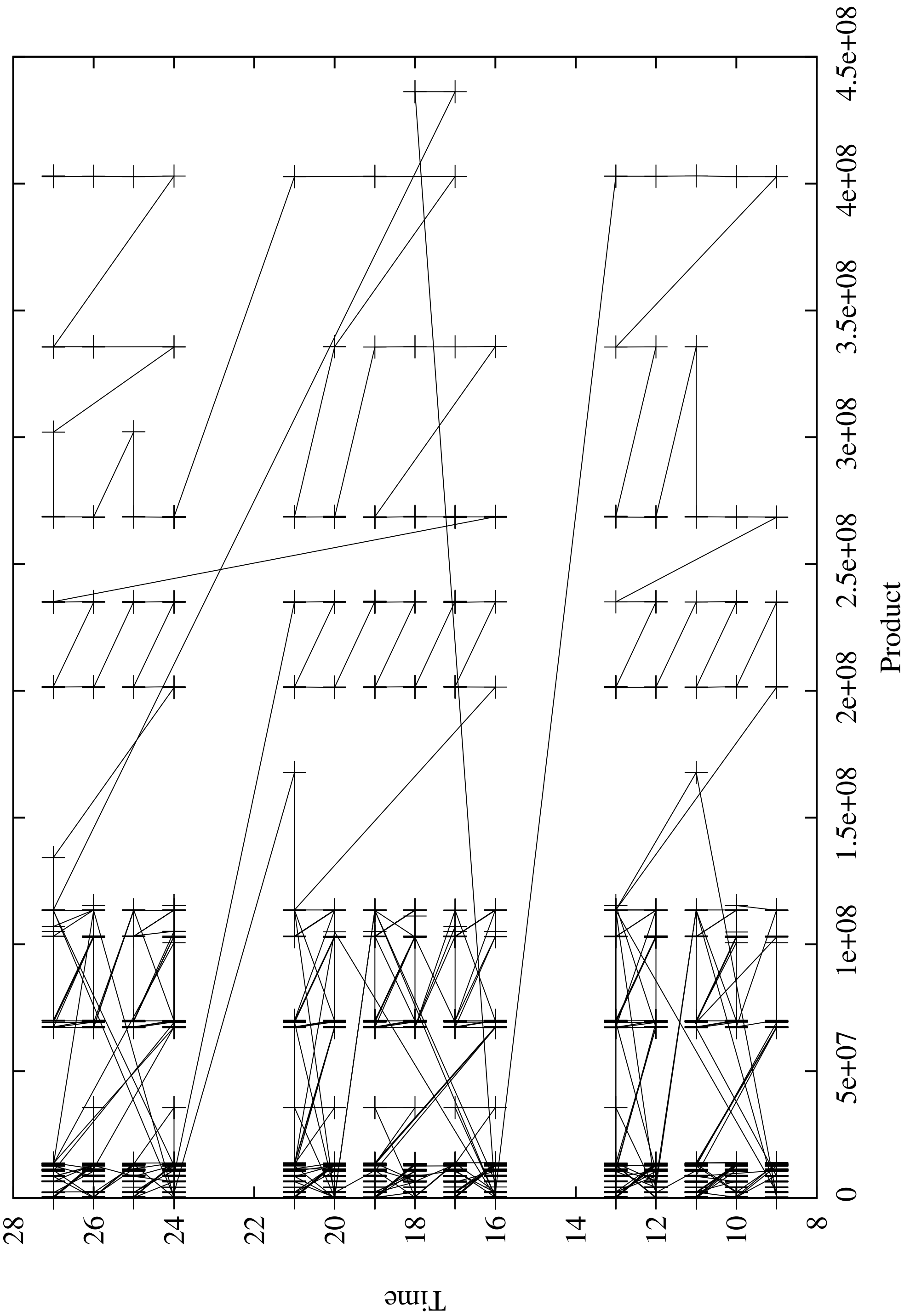


Figure B.6: GfK Data Distribution: 2d Projection of (Product,Time) with points connected by a line in Z-order.

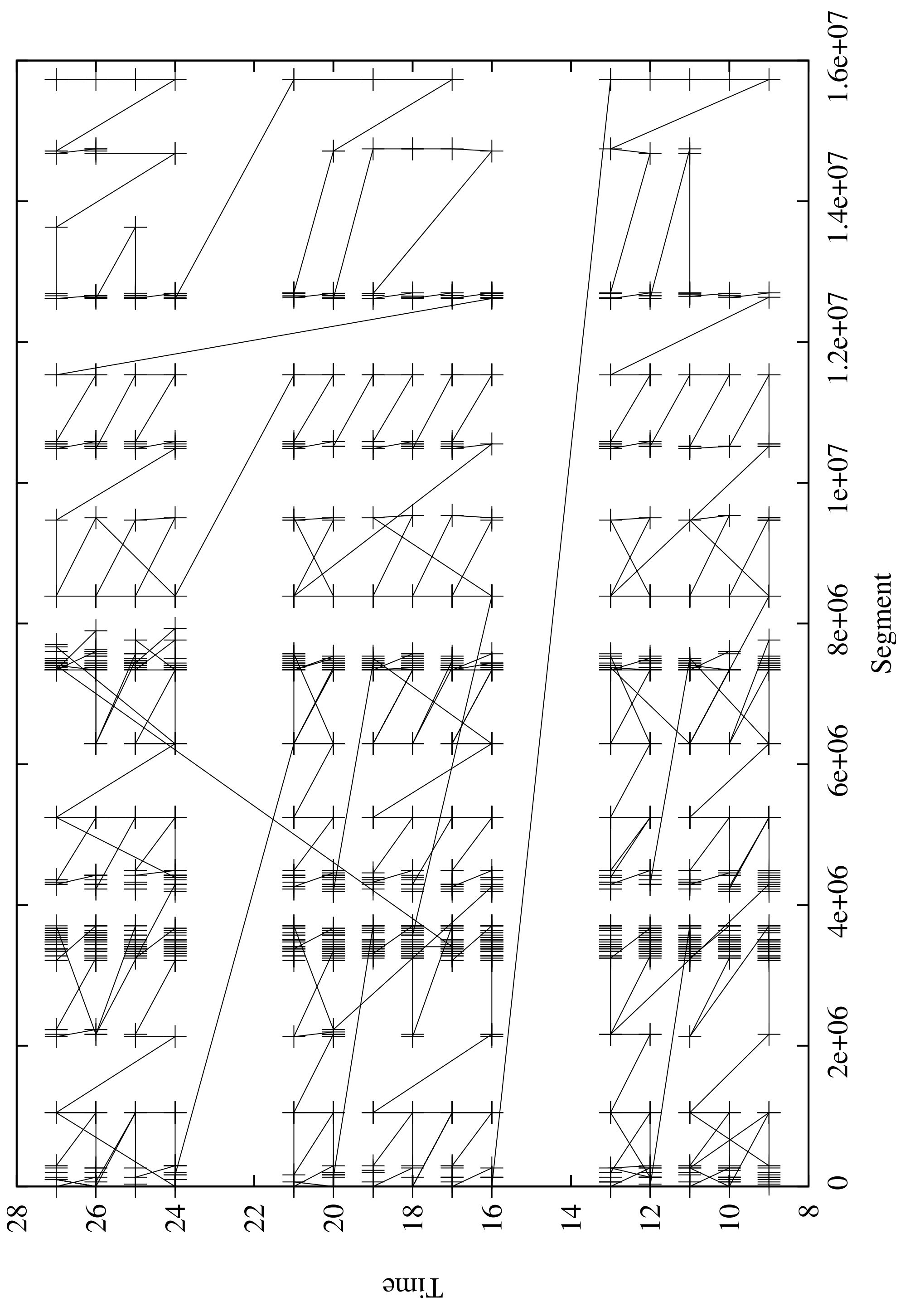


Figure B.7: GfK Data Distribution: 2d Projection of (Segment,Time) with points connected by a line in Z-order.

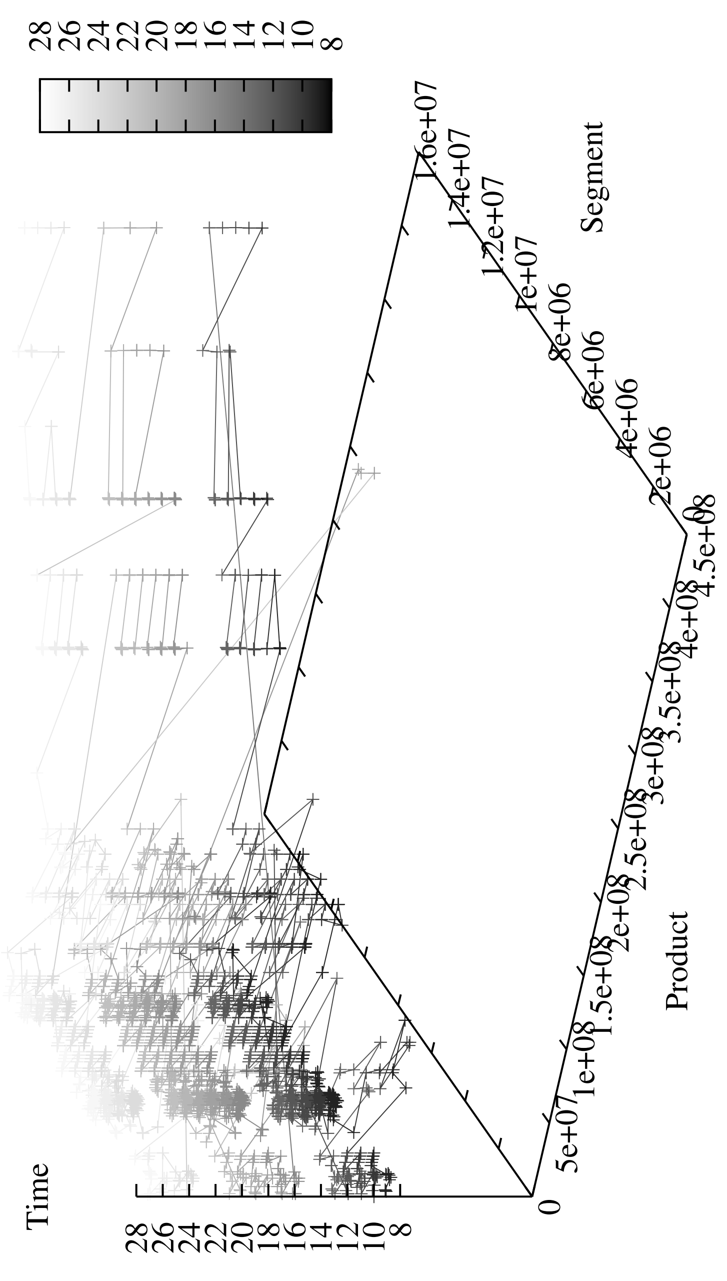


Figure B.8: GfK Data Distribution: 3d
with points connected in Z-order by a line colored by a gradient from black to white.

B.2 GEO Data

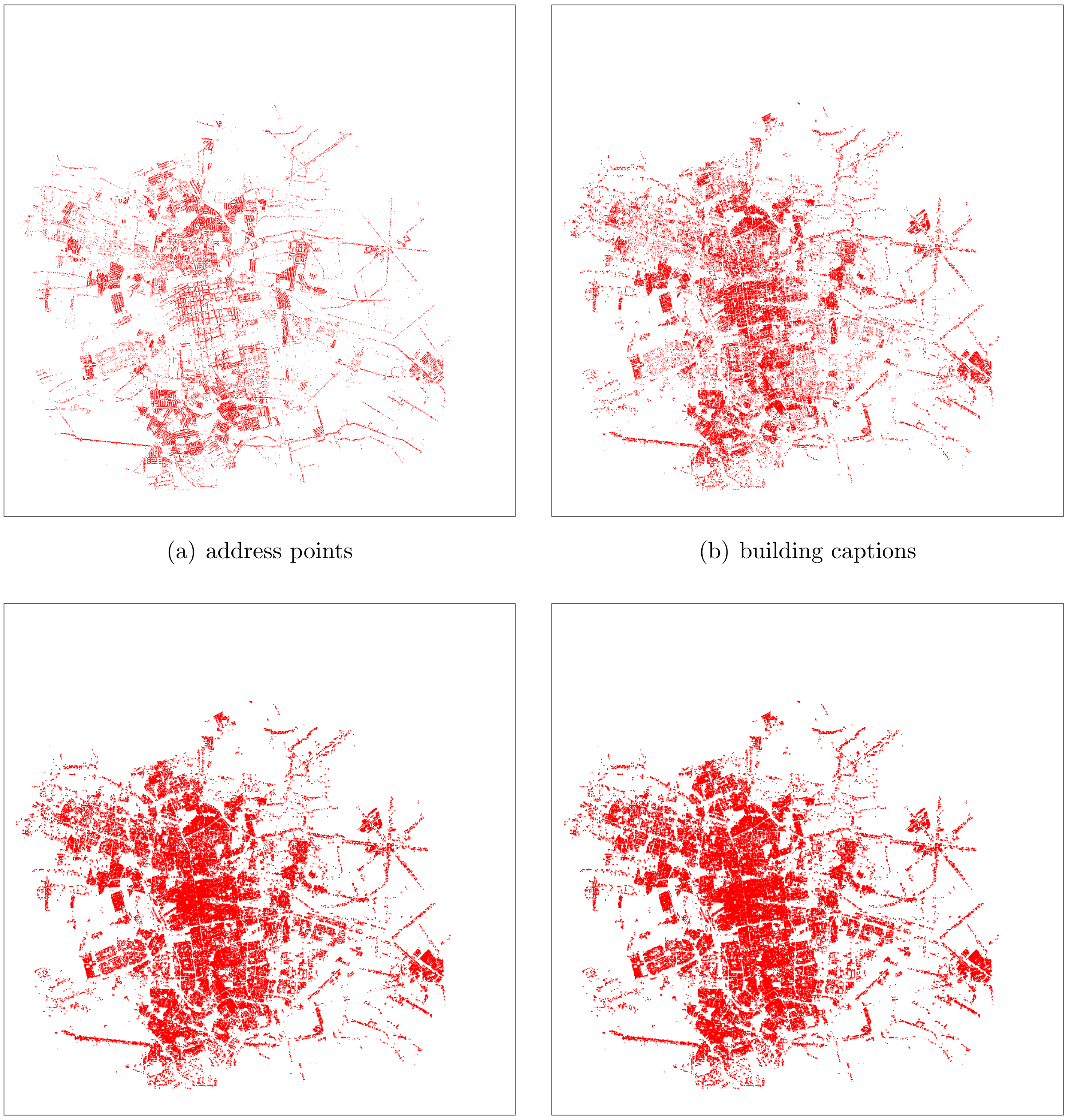
The spatial real world data used fro measurements is a data set describing spatial properties of a Polish city. Buildings, streets, power lines and other spatial objects are part of this data set. The exact numbers and types of spatial objects are given in the following table:

Object Count	**Type**	**Description**
7404	point	street section captions
7406	open polygon	street sections lines
16957	open polygon	power line sections
54062	point	address point
69764	closed polygon	plot area
69764	point	plot caption
144136	point	building caption
144155	closed polygon	building area
$\sum = 513648$	-	-

The domain of the coordinates is $[4M, 6.5M]$ for both dimensions.

Plots of the data are in Figure B.9, Figure B.10, and Figure B.11. Buildings are relatively small and their MBBs are tight, i.e., the polygons are good approximated by the MBBs.

The power line and street sections are not approximated well by their MBBs if they are not aligned to one of the axis.



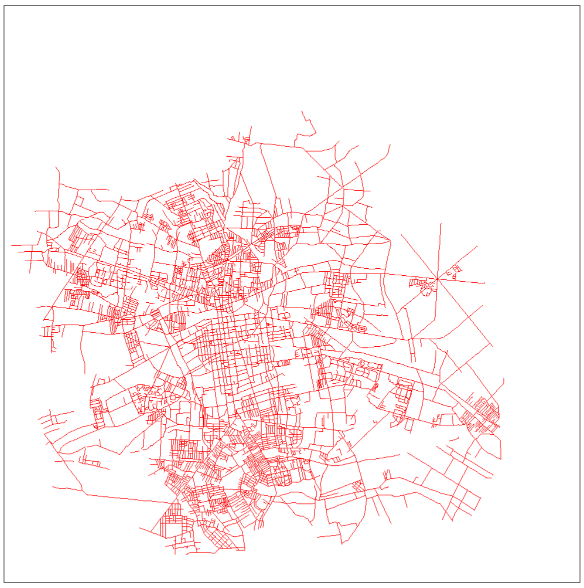
(a) address points

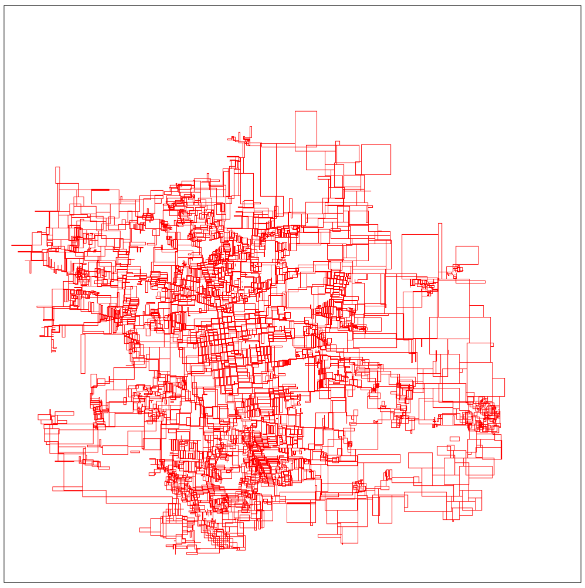
(b) building captions

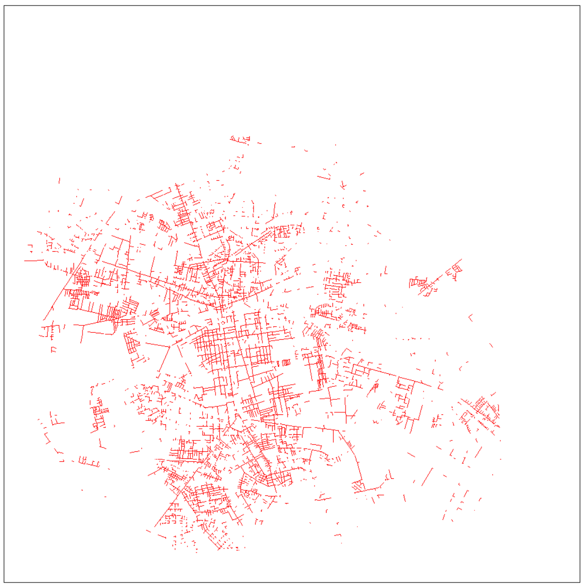
(c) building areas

(d) building area MBBs

Figure B.9: GEO Data: Addresses and Buildings


(a) street sections


(b) street section MBBs


(c) street section captions

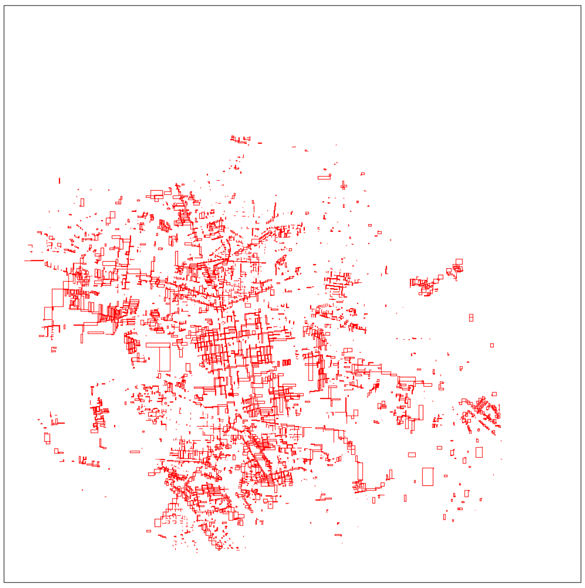
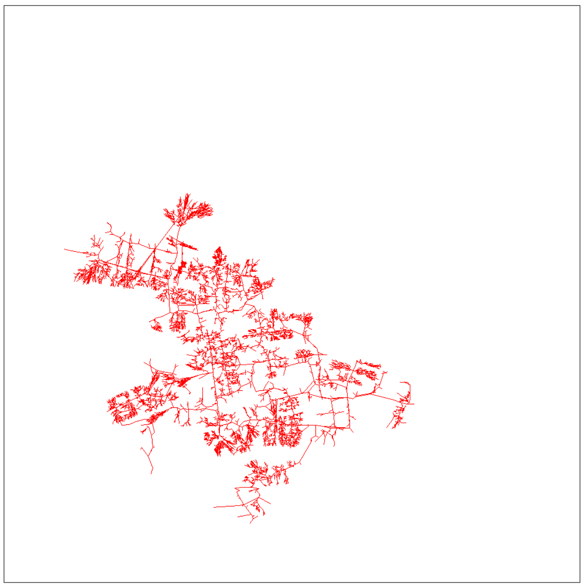
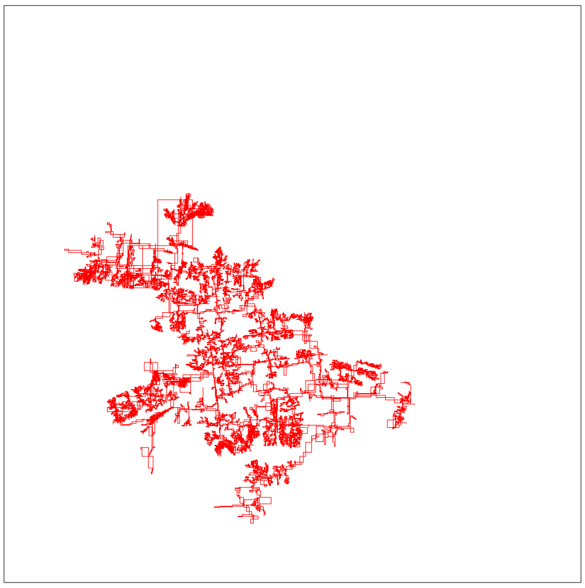

(d) street section caption MBBs

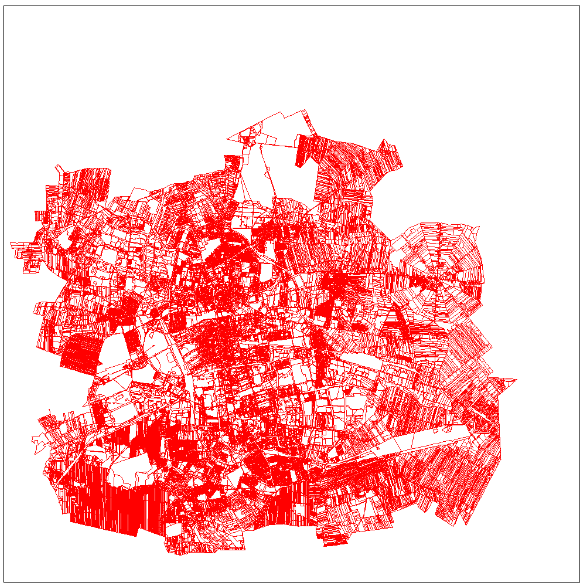
Figure B.10: GEO Data: Streets and Street Captions



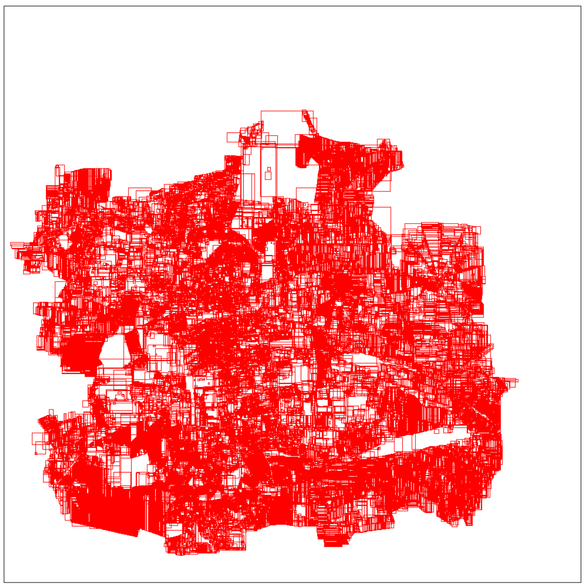
(a) power line section



(b) power line section MBBs



(c) plot area



(d) plot area MBBs

Figure B.11: GEO Data: Power Lines and Plot Area

Appendix C

Measures of RFDBMS

t	number of result tuples
ft	number of tuples from data pages which are no result (false) tuples
time	overal time for query execution in msec (1sec == 1000msec)
pR	physical page reads
lR	logical page reads
dR	delayed page reads
pIR	physical index page reads
lIR	logical index page reads
pDR	physical data page reads
lDR	logical data page reads
fDR	false data page reads, i.e. pages contributing no result tuple
pW	physical page writes
lW	logical page writes
dR	delayed page writes
pIW	physical index page writes
lIW	logical index page writes
pDW	physical data page writes
lDW	logical data page writes
tr	number of tuples read from index/data pages
dist	number of distance calculations
ps	page size in bytes
Ih	index height (only valid for tree based indexes)
Ic	capacity of an index page
It	number of index tuples
Ip	number of index pages
Iu	average page utilization of index pages
Dc	capacity of a data page
Dt	number of data tuples
Dp	number of data pages
Du	average page utilization of data pages

Bibliography

[ABH$^+$04] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. The Priority R-Tree: A Practical Efficient and Worst-Case Optimal R-Tree. In *Proc. of SIGMOD, Paris, France*, June 2004.

[Ada80] Douglas Adams. *The Hitchhiker's Guide To The Galaxy.* Crown Publishing Group, 1980.

[Agg00] Alok Aggarwal. External Memory Algorithms and Data Structures: Dealing with MASSIVE DATA. In *Draft Version, February 4*. http://www.cs.duke.edu/ jsv/Papers/catalog/, 2000.

[AH85] J.F. Allen and P.J. Hayes. A common-sense theory of time. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, volume 9., pages 528–531, 1985.

[Ant94] A. Antoshenkov. Byte-aligned bitmap compression. U.s. patent number5,363,098, Oracle Corp., 1994.

[Aok98] Paul M. Aoki. Generalizing “search” in generalized search trees (extended abstract). In *Proceedings of the Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA*, pages 380–389. IEEE Computer Society, 1998.

[Bau98a] Michael Bauer. Implementation and evaluation of variable ub-trees. Master's thesis, Technische Universität München, August 1998.

[Bau98b] Michael Bauer. A mass-loading tool for the ub-tree. Technical report, Technische Universität München, 1998.

[Bay96] Rudolf Bayer. The universal B-Tree for multidimensional Indexing. Technical Report TUM-I9637, Institut für Informatik, TU München, 1996.

[Bay97] Rudolf Bayer. The universal B-Tree for multidimensional Indexing: General Concepts. In *World-Wide Computing and its Applications '97 (WWCA '97), Lecture Notes on Computer Science.* Springer Verlag, 1997. Tsukuba, Japan.

[Ben75] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[Ben79] J. L. Bentley. Multidimensional binary search in database applications. *IEEE Transactions on Software Engineering*, 4(5):333–340, 1979.

[BG94] Gabriele Blankenagel and Ralf Hartmut Güting. External segment trees. In *Algorithmica*, volume 12(6), pages 498–532, 1994.

[BGS02] Tom Barclay, Jim Gray, and Don Slutz. *Microsoft TerraServer: A Spatial Data Warehouse*, 2002.

[BKK99] Christian Böhm, Gerald Klump, and Hans-Peter Kriegel. XZ-Ordering: A space-filling curve for objects with spatial extension. In *Proceedings of Advances in Spatial Databases, 6th International Symposium, SSD'99, Hong Kong, China, July 20-23, 1999*, volume 1651 of *Lecture Notes in Computer Science*, pages 75–90. Springer, 1999.

[BKS$^+$90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 322–331. ACM Press, 1990.

[BM72] Rudolf Bayer and E. McCreight. Organization and Maintainance of large ordered Indexes. In *Acta Informatica 1*, pages 173–189, 1972.

[BM97] Rudolf Bayer and Volker Markl. The UB-Tree: Performance of Multidimensional Range Queries. Technical Report TUM-I9814, Institut für Informatik, TU München, 1997.

[Boz98] Toga Bozkaya. *Index Structures For Temporal And Multimedia Databases.* PhD thesis, Department of Computer Engineering and Science Case Western Reserve University, 1998.

[BPT02] Sotiris Brakatsoulas, Dieter Pfoser, and Yannis Theodoridis. Revisiting r-tree construction principles. In *Advances in Databases and Information Systems : 6th East European Conference, ADBIS 2002, Bratislava, Slovakia*, page 149, 2002.

[BRB03] Michael G. Bauer, Frank Ramsak, and Rudolf Bayer. Multidimensional Mapping and Indexing of XML. In *BTW 2003, Datenbanksysteme für Business, Technologie und Web, Tagungsband der 10. BTW-Konferenz, 26.-28. Februar 2003, Leipzig*, volume 26 of *LNI*, pages 305–323. GI, 2003.

[But71] Arthur R. Butz. An alternative algorithm for hilbert's space-filling curve. *IEEE Transactions on Computers*, 20:424–426, April 1971.

[BY89] Ricardo A. Baeza-Yates. The Expected Behaviour of B^+-Trees. In *Acta Informatica 26(5)*, pages 439–471, 1989.

[CCF$^+$99] Weidong Chen, Jyh-Herng Chow, You-Chin Fuh, Jean Grandbois, Michelle Jou, Nelson Mendonça Mattos, Brian T. Tran, and Yun Wang. High level indexing of user-defined types. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 554–564. Morgan Kaufmann, 1999.

[CI98] Chee Yong Chan and Yannis E. Ioannidis. Bitmap index design and evaluation. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 355–366. ACM Press, 1998.

[CI99] Chee Yong Chan and Yannis E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 215–226. ACM Press, 1999.

[CM99] Surajit Chaudhuri and Rajeev Motwani. On sampling and relational operators. In *IEEE Data Engineering Bulletin, 22(4)*, pages 41–46, 1999.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6):377–387, 1970.

[Com79] Douglas Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[Ede95] Herb Edelstein. Technology analysis: Faster data warehouses. *Information Week*, December 1995.

[Fal86] Christos Faloutsos. Multiattribute hashing using gray codes. In Carlo Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 28-30, 1986*, pages 227–238. ACM Press, 1986.

[Fen00] Robert Fenk. Management and query processing of one-dimensional intervals with the UB-tree. In *EDBT PhD Workshop*, 2000.

[Fen02] Robert Fenk. The BUB-Tree. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, 2002, Hong Kong, China*, 2002. Poster Session.

[FFS00] Johann Christoph Freytag, M. Flasza, and Michael Stillger. Implementing geospatial operations in an object-relational database system. In *SSDBM*, pages 209–219, 2000.

[FKM$^+$00] R. Fenk, A. Kawakami, V. Markl, R. Bayer, and S. Osaki. Bulk loading a data warehouse built upon a ub-tree. In *Proc. of IDEAS Conference*, Yokohama, Japan, 2000.

[FMB99] Robert Fenk, Volker Markl, and Rudolf Bayer. Improving multidimensional range queries of non rectangular volumes specified by a query box set. In *Proc. of International Symposium on Database, Web and Cooperative Systems (DWACOS)*, Baden-Baden, Germany, 1999.

[FMB02] Robert Fenk, Volker Markl, and Rudolf Bayer. Inverval Processing with the UB-Tree. In *Proc. of IDEAS Conf., Edmonton, Canada*, 2002.

[FNP$^+$79] Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible hashing - a fast access method for dynamic files. *TODS*, 4(3):315–344, 1979.

[FR89] Christos Faloutsos and Shari Roseman. Fractals for secondary key retrieval. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 29-31, 1989, Philadelphia, Pennsylvania*, pages 247–252. ACM Press, 1989.

[FR91] Christos Faloutsos and Yi Rong. Dot: A spatial access method using fractals. In *Proceedings of the Seventh International Conference on Data Engineering, April 8-12, 1991, Kobe, Japan*, pages 152–159. IEEE Computer Society, 1991.

[Fri97] Nils Frielinghaus. Evaluierung der einsatzfähigkeit des ub–baumes für das sap–system r/3. Master's thesis, TUM, November 97.

[Gae95] Volker Gaede. Optimal redundancy in spatial database systems. In *Symposium on Large Spatial Databases*, pages 96–116, 1995.

[GG98] Volker Gaede and Oliver Günther. Multidimensional Access Methods. In *Computing Surveys 30(2)*, pages 170–231. ACM Press, 1998.

[GKK$^+$01] Andreas Gärtner, Alfons Kemper, Donald Kossmann, and Bernhard Zeller. Efficient bulk deletes in relational databases. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, pages 183–192. IEEE Computer Society, 2001.

[GLL98] Yván J. García, Mario A. Lopez, and Scott T. Leutenegger. A Greedy Algorithm for Bulk Loading R-Trees. In *ACM International Workshop on Advances in Geographic Information Systems*, pages 163–164, 1998.

[GLO$^+$96] Cheng Hian Goh, Hongjun Lu, Beng Chin Ooi, and Kian-Lee Tan. Indexing temporal data using existing b+-trees. In *Data & Knowledge Engineering*, volume 18(2), pages 147–165, 1996.

[Gra47] Frank Gray. *Pulse Code Communications*. Bell Telephone Laboratories, us patent 2632058 edition, 11 1947.

[Gra03a] Goetz Graefe. Partitioned B-Trees - a user's guide. In *BTW: Leipzig*, volume 10, pages 668–671, 2003.

[Gra03b] Goetz Graefe. Sorting and Indexing with Partitioned B-Trees. In *First Biennial Conference on Innovative Data Systems Research (CIDR): Asilomar, CA, USA*, volume 1, pages 668–671, 2003.

[Gra04] Goetz Graefe. Write-Optimized B-Trees. In *VLDB: Toronto, Canada*, volume 30, 2004.

[Gui03] Toms Hardware Guide. Hard drives instead of tapes. http://slashdot.org/articles/03/04/24/1921205.shtml?tid=126 http://www.tomshardware.com/storage/20030425/, 2003.

[Gut84] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In Beatrice Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57. ACM Press, 1984.

[Hil91] David Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38:459–460, 1891.

[HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 562–573. Morgan Kaufmann, 1995.

[Hod03] Chris Hodges. Approximation of arbitrary polygonal objects using space filling curves versus a bounding box approach. Master's thesis, 2003. Systementwicklungs Projekt.

[HSS97] Gisli R. Hjaltason, Hanan Samet, and Yoram J. Sussmann. Speeding up Bulk-Loading of Quadtrees. In *ACM International Workshop on Advances in Geographic Information Systems*, pages 50–53, 1997.

[HSW89] Andreas Henrich, Has-Werner Six, and Peter Widmayer. The LSD tree: spatial access to multidimensional point and non point objects. In *In Proc. VLDB*, pages 45–53, 89.

[HW02] Jan Hungershöfer and Jens-Michael Wierum. On the quality of partitions based on space-filling curves. In LNCS 2331, editor, *International Conference on Computational Science*, pages 36–45, 2002.

[IBM01] IBM. *IBM Ultrastar 36Z15 hard disk drive data sheet*, 2001.

[IBM02] IBM. *IBM DB2 Spatial Extender*, 2002.

[Inc97] Sybase Inc. *Sybase IQ Indexes. Sybase IQ Administration Guide*, sybase iq, release 11.2 collection, chapter 5 edition, March 1997.

[Inf99] Informix. *Informix Dynamic Server with Universal Data Option, Version 9.1.X, Documentation.* Informix, 1999.

[Jag90a] H. V. Jagadish. Linear clustering of objects with multiple attributes. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 332–342. ACM Press, 1990.

[Jag90b] H. V. Jagadish. Spatial search with polyhedra. In *Proceedings of the Sixth International Conference on Data Engineering, February 5-9, 1990, Los Angeles, California, USA*, pages 311–319. IEEE Computer Society, 1990.

[Jan95] Jan Jannink. Implementing deletion in b+-trees. *SIGMOD Record*, 24(1):33–38, 1995.

[Joh99] Theodore Johnson. Performance measurements of compressed bitmap indices. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 278–289. Morgan Kaufmann, 1999.

[jT03] java.XXL Team. java.xxl: extensible and flexible library. http://dbs.mathematik.uni-marburg.de/research/projects/xxl/, 2003.

[Kau91] M. Kaufmann. *Pearl, Judea: Probabilistic reasoning in intelligent systems: networks of plausible inference / Judea Pearl.* Rev. 2. print. - San Mateo, Calif., 1991.

[KB95] Marcel Kornacker and Douglas Banks. High-concurrency locking in rtrees. In *In Proc. of VLDB, Zürich, Switzerland*, 1995.

[KF93] Ibrahim Kamel and Christos Faloutsos. On Packing R-trees. In *CIKM*, pages 490–499, 1993.

[KKaS97] T. Kahabka, M. Korkea-aho, and G. Specht. Gras: An adaptive personalization scheme for hypermedia databases. In *Proc. of the 2nd. Conference on Hypertext - Information Retrieval - Multimedia (HIM '97)*, pages 279 – 292, 1997.

[KMH97] Marcel Kornacker, C. Mohan, and Joseph M. Hellerstein. Concurrency and recovery in generalize search trees. In *In Proc. of SIGMOD*, 1997.

[KMP$^+$01] Hans-Peter Kriegel, Andreas Müller, Marco Pötke, and Thomas Seidl. Spatial data management for computer aided design. In *Proceedings of SIGMOD'01, Santa Barbara*. ACM Press, 2001.

[Knu73] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison - Wesley, 1973.

[Kou00] Nick Koudas. Space efficient bitmap indexing. In *Proceedings of the ninth international conference on Information and knowledge management*, pages 194–201. ACM Press, 2000.

[KPS00] Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl. Managing intervals efficiently in object-relational databases. In *Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 407–418. Morgan Kaufmann, 2000.

[KPS01] H.-P. Kriegel, M. Pötke, and T. Seidl. Interval Sequences: An Object-Relational Approach to Manage Spatial Data. In *Proc. 7th Int. Symp. on Spatial and Temporal Databases (SSTD'01), Redondo Beach, CA, 2001.*, 2001.

[KRR02] Donald Kossmann, Frank Ramsak, and Steffen Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, 2002, Hong Kong, China*, pages 275–286, 2002.

[KS91] Curtis P. Kolovson and Michael Stonebraker. Segment indexes: Dynamic indexing techniques for multi-dimensional interval data. In James Clifford and Roger King, editors, *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado, May 29-31, 1991*, pages 138–147. ACM Press, 1991.

[KTS$^+$02] Nikos Karayannidis, Aris Tsois, Timos K. Sellis, Roland Pieringer, Volker Markl, Frank Ramsak, Robert Fenk, Klaus Elhardt, and Rudolf Bayer. Processing Star Queries on Hierarchically-Clustered Fact Tables. In *28. VLDB: Hong Kong, China*, pages 730–741, 2002.

[Küs83] Klaus Küspert. Storage Utilization in B*-Trees with a Generalized Overflow Technique. In *Acta Informatica 19*, pages 35–55, 1983.

[Law00] Jonathan Lawder. *The Application of Space-filling Curves to the Storage and Retrieval of Multi-dimensional Data.* PhD thesis, University of London, 2000.

[LEL97] Scott T. Leutenegger, J. M. Edgington, and Mario A. Lopez. STR: A Simple and Efficient Algorithm for R-Tree Packing. In Alex Gray and Per-Åke Larson, editors, *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K*, pages 497–506. IEEE Computer Society, 1997.

[LN97] Scott T. Leutenegger and David M. Nicol. Efficient Bulk-Loading of Gridfiles. In *Proceedings of the IEEE Transactions on Knowledge and Data Engineering, Volume 9(3)*, pages 410–420, 1997.

[LR95] Ming-Ling Lo and Chinya V. Ravishankar. Generating Seeded Trees from Data Sets. In *Symposium on Large Spatial Databases*, pages 328–347, 1995.

[MAK02] Mohamed F. Mokbel, Walid G. Aref, and Ibrahim Kamel. Performance of multi-dimensional space-filling curves. In ACM GIS, editor, *In Proceedings of the International Conference on Geographic Information System*, Nov 2002.

[Mar99] Volker Markl. *Processing Relational Queries using a Multidimensional Access Technique.* PhD thesis, DISDBIS, Band 59, Infix Verlag, 1999.

[Mer99] Stefan Merkel. Evaluation of the ub-tree for a market research data warehouse. Master's thesis, Technische Universität München, August 1999.

[Mic00] Microsoft. *SQL Server Books Online, Microsoft Foundation.* Microsoft, 2000.

[MJF$^+$01] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *Knowledge and Data Engineering*, 13(1):124–141, 2001.

[MO95] R. Maelbrancke and H. Olivie. Optimizing jan jannink's implementation of b+-tree deletion. *SIGMOD Record*, 24(3):5–7, 1995.

[Moo00] Eliakim Hastings Moore. On certain crinkly curves. In *Transactions of the American Mathematical Society*, volume 1, pages 72–90, Jan 1900.

[Moo00] Doug Moore. hilbert.c, Feb 2000.

[Mor66] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., Ottawa, Canada, 1966.

[MRB99] Volker Markl, Frank Ramsak, and Rudolf Bayer. Improving OLAP Performance by Multidimensional Hierarchical Clustering. In *Proc. of IDEAS Conf., Montreal, Canada*, 1999.

[MTT00] Y. Manolopoulos, Y. Theodoridis, and V.J. Tsotras. Chapter 4: Access methods for intervals. In *Advanced Database Indexing*, Boston, MA: Kluwer, 2000.

[MZB99] Volker Markl, Martin Zirkel, and Rudolf Bayer. Processing Operations with Restrictions in RDBMS without External Sorting: The Tetris Algorithm. In *Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Austrialia*, pages 562–571. IEEE Computer Society, 1999.

[NHS84] Jürg Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *TODS*, 9(1):38–71, 1984.

[Nic01] Oliver Nickel. Processing Two-Dimensional Extended Objects with the UB-Tree. Master's thesis, Technische Universität München, 2001. Systementwicklungs Projekt.

[NK94] Vincent Ng and Tiko Kameda. The R-Link Tree: A Recoverable Index Structure for Spatial Data. In *In Proc. of DEXA*, 1994.

[OM84] Jack A. Orenstein and T. H. Merrett. A Class of Data Structures for Associative Searching. In *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, April 2-4, 1984, Waterloo, Ontario, Canada*, pages 181–190. ACM, 1984.

[O'N89] Patrick E. O'Neil, editor. *Model 204 Architecture and Performance*, volume 359 of *Lecture Notes in Computer Science*. Springer, 1989.

[O'N97] P. O'Neil. Informix and indexing support for data warehouses, 1997.

[Ooi87] Beng Chin Ooi. Spatial kd-tree: A data structure for geographic database. In *BTW*, pages 247–258, 1987.

[Ooi90] Beng Chin Ooi. Efficient Query Processing in Geographic Information Systems. In *Number 471 in LNCS*. Springer Verlag, 1990.

[OQ97] Patrick E. O'Neil and Dallan Quass. Improved query performance with variant indexes. In Joan Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 38–49. ACM Press, 1997.

[Ora99] Oracle. *Oracle 8i Utilities / Documentation.* Oracle Corporation, 1999.

[Ora00a] Oracle. *Oracle8i Data Cartridge Developers's Guide.* Oracle Corporation, Redwood City, CA, rel. 8.1.7 edition, 2000.

[Ora00b] Oracle. *Oracle8i Documentation.* Oracle Corporation, Redwood City, CA, rel. 8.1.7 edition, 2000.

[Ora00c] Oracle. *Oracle8i Spatial User's Guide and Reference.* Oracle Corporation, Redwood City, CA, rel. 8.1.7 edition, 2000.

[Ore90] Jack A. Orenstein. A Comparison of Spatial Query Processing Techniques for Native and Parameter Spaces. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 343–352. ACM Press, 1990.

[OS90] Y. Ohsawa and M. Sakauchi. Bd-tree: A new tre type data structure with homogeneous nodes suitable for a very large spatial database. In *Proc. 6th ICDE*, pages 296–303, 1990.

[Pal00] Alan Paller. Rely on Red Brick's Performance for Data Warehouse Applications. Technical report, Data Warehousing Institute, Informix Corporation, 2000.

[Pea90] Giuseppe Peano. Sur une courbe qui remplit toute une air plaine. *Mathematische Annalen*, 36:157–160, 1890.

[PER$^+$03a] Roland Pieringer, Klaus Elhardt, Frank Ramsak, Volker Markl, Robert Fenk, Robert Bayer, Nikos Karayannidis, Aris Tsois, and Timos K. Sellis. Combining Hierarchy Encoding and Pre-Grouping: Intelligent Grouping in Star Join Processing. In *19. ICDE: Bangalore, India*, pages 329–340, 2003.

[PER$^+$03b] Roland Pieringer, Klaus Elhardt, Frank Ramsak, Volker Markl, Robert Fenk, and Rudolf Bayer. Transbase: a Leading-edge ROLAP Engine Supporting Multidimensional Indexing and Hierarchy Clustering. In *10. BTW: Leipzig*, pages 648–667, 2003.

[Pie98] Roland Pieringer. Evaluation of the ub-tree in the sap environment. Master's thesis, Technische Universität München, November 1998.

[Pie03] Roland Pieringer. *Modeling and implementing multidimensional hierarchically structured Data for Data Warehouses in relational Database Management systems and the Implementation onto Transbase.* PhD thesis, Technische Universität München, 2003.

[PKF00] Bernd-Uwe Pagel, Flip Korn, and Christos Faloutsos. Deflating the dimensionality curse using multiple fractal dimensions. In *ICDE*, pages 589–598, 2000.

[Ram97] S. Ramaswamy. Efficient indexing for constraint and temporal databases. In *Proceedings of the 6th International Conference on Database Theory (ICDT)*, pages 419–431, 1997.

[Ram02] Frank Ramsak. *Towards a general-purpose, multidimensional index: Integration, Optimization, and Enhancement of UB-Trees.* PhD thesis, Technische Universität München, 2002.

[RL85] Nick Roussopoulos and Daniel Leifker. Direct Spatial Search on Pictorial Databases Using Packed R-Trees. In Shamkant B. Navathe, editor, *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, May 28-31, 1985*, pages 17–31. ACM Press, 1985.

[RMF$^+$00] Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhard, and Rudolf Bayer. Integrating the UB-Tree into a Database System Kernel. In *Proceedings of International Conference on Very Large Data Bases, 2000, Cairo, Egypt*, 2000.

[RMF$^+$01] F. Ramsak, V. Markl, R. Fenk, R. Bayer, and T. Ruf. Interactive ROLAP on Large Databases: A Case Study with UB-Trees. In *Proc. of IDEAS Conf. 2001, Grenoble, France*, 2001.

[RS81] Arnold L. Rosenberg and Lawrence Snyder. Time- and Space-Optimality in B-Trees. *TODS*, 6(1):174–193, 1981.

[Sag94] Hans Sagan. Space-filling curves. *Springer-Verlag*, 1994.

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison Wesley, 1990.

[Sch93] M. Schiwietz. *Speicherung und Anfragebearbeitung komplexer Geo-Objekte.* PhD thesis, Ludwig-Maximilians-Universität München, Germany, 1993. In German.

[Sea01] Seagate. *Cheetah 73 Family: ST173404LW/LWV/LC/LCV*, product manual volume 1, ref f edition, 1 2001.

[SK88] B. Seeger and H.-P. Kriegel. Techniques for design and implementation of efficient spatial access methods. In *In Proc. VLDB*, pages 360–371, 1988.

[Sma02] Smallworld. *Smallworld GIS Database*, 2002.

[SOL94] Han Shen, Beng Chin Ooi, and Hongjun Lu. The tp-index: A dynamic and efficient indexing mechanism for temporal databases. In *Proceedings of the Tenth International Conference on Data Engineering, February 14-18, 1994, Houston, Texas, USA*, pages 274–281. IEEE Computer Society, 1994.

[SRF87] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *VLDB '87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 507–518. Morgan Kaufmann, 1987.

[TAS00] TAS. *TransBase HyperCube.* TransAction Software, http://www.transaction.de, 2000.

[TCG+93] A.U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Sondgrass. *Temporal Databases: Theory, Design and Implementation.* Benjamin/Cummings, Redwood City, CA, 1993.

[Tea00] Shore Team. Shore - a high-performance, scalable, persistent object repository (version 2.0). http://www.cs.wisc.edu/shore/, 2000.

[Tea03a] PostgreSQL Team. Postgresql version 7.3.4. http://www.postgresql.org/, 2003.

[Tea03b] Predator Team. Predator - enhanced data type object-relational dbms. http://www.distlab.dk/predator, 2003.

[TH81] Hermann Tropf and H. Herzog. Multidimensional range search in dynamically balanced trees. *Agewandte Informatik*, pages 71–77, 2 1981. Download via http://www/vision-tools.com/h-tropf.

[Tra99] Transaction Processing Performance Council (TPC). *TCP Benchmark H*, decision support, standard specification, revision 1.3.0 edition, 1999.

[Tro03] Hermann Tropf. Efficient processing of multidimensional databases with data sorted in hilbert order. In *Submitted to VLDB'2003, Berlin, Germany*, 2003.

[Unt02] Werner Unterhofer. A Comparison of Mapping Methods for Extended Objects. Master's thesis, Technische Universität München, 2002. Systementwicklungs Projekt.

[WB98] Ming-Chuan Wu and Alejandro P. Buchmann. Encoded bitmap indexing for data warehouses. In *Proceedings of the Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA*, pages 220–230. IEEE Computer Society, 1998.

[Wel84] Terry A. Welch. A technique for high performance data compression. *Computer Magazine of the Computer Group News of the IEEE Computer Group Society*, 17(6):8–19, June 1984.

[Wid04] Robert Widhopf. AFFIC: A foundation for index comparisons. In *EDBT 2004, 2004, Heraklion, Greece*, 2004. Demo Session.

[Win99] R. Winter. Indexing goes a new direction. *Intelligent Enterprise*, 2(2):70–73, January 199.

[Zir03] Martin Zirkel. *Bewertung des UB-Baums in der Anfrageverarbeitung unter Berücksichtigung der Sortierung.* PhD thesis, Technische Universität München, 2003.

[ZS96] Chendong Zou and Betty Salzberg. On-line reorganization of sparsely-populated b+trees. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 115–124. ACM Press, 1996.