Charles University in Prague

Faculty of Mathematics and Physics

DIPLOMA THESIS

CHARLES UNIVERSITY PRAGUE

faculty of mathematics and physics

Antonín Prukl

A RELATIONAL APPROACH TO INDEXING

RELAČNÍ PŘÍSTUP K INDEXACI

Department of Software Engineering

Supervisor: Prof. RNDr. Jaroslav Pokorný, CSc.

Study Program: Computer Science

I would like to thank my supervisor, Prof. RNDr. Jaroslav Pokorný, CSc., for his valuable advice.

I hereby declare that I have written this diploma thesis on my own, and used no other than the named sources and aids. I agree with lending the thesis.

Prague, April 20, 2007 Antonín Prukl

Contents

Název práce: Relační přístup k indexaci

Autor: Antonín Prukl

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: Prof. RNDr. Jaroslav Pokorný, CSc.

e-mail vedoucího: jaroslav.pokorny@mff.cuni.cz

Abstrakt: Za účelem efektivního vyhodnocení SQL dotazů mohou uživatelé databázových systému využít řadu specializovaných přístupových metod, které se obecně nazývají indexy.

V některých případech však nemusí být množina metod poskytovaná databázovým systémem dostačující. Jednou z možností, jak implementovat nový index v relačním SŘBD, je využít tabulek daného systému. Tento přístup nevyžaduje změny v jádru databázového systému a je tak dostupný vývojářům i v případě, že cílový SŘBD není distribuován jako open source. V rozšiřitelné databázové architektuře je tak vyžadována pouze možnost přidat nový datový typ do stávajícího SŘBD.

V této práci byl zmíněným způsobem integrován UB-strom do SŘBD Oracle. Relační tabulky související s indexem byly navrženy dvěma různými způsoby, zároveň byly zkoumány čtyři metody pro vyhodnocení relevantních SQL dotazů. V rámci experimentů bylo pak implementované řešení relačního indexu porovnáno s nativním nasazením téhož indexu.

Klíčová slova: databázové systémy, relační indexace, UB-strom, benchmarkování

Title: A Relational Approach to Indexing

Author: Antonín Prukl

Department: Department of Software Engineering

Supervisor: Prof. RNDr. Jaroslav Pokorný, CSc.

Supervisor's e-mail address: jaroslav.pokorny@mff.cuni.cz

Abstract: In order to achieve efficient evaluation of SQL queries, database systems provide its users with set of integrated index access methods.

When a new access method is required for various reasons, one of the possibilities to implement such method in a relational DBMS is the way of exploiting relational tables of given database system. This approach does not involve any internal changes of database system kernel and thus it is available to all developers even when the target DBMS is not distributed as an open source. In the terms of extensible database architecture, only the availability to extend existing DBMS with a new data type is required.

In this work, UB-Tree index has been integrated into Oracle DBMS in such way. Index related tables have been designed in two different ways and four alternatives to evaluate relevant queries have been proposed and studied. Finally, several experiments have been done to compare performance of an access method implemented via the relational approach and a native kernel integration of the same method.

Keywords: database systems, relational indexing, UB-Tree, benchmarking

CHAPTER 1

Introduction

In database area, working with high amount of data often brings a requirement for speeding up the access time to searched entries with usage of some specialized secondary data structures. This is obvious mainly in case when the amount of searched data is very small in comparison with the volume of all data. Such structures that are used for direct access to a small subset of data instead of sequential passing through all data are called indexes.

Currently database systems provide a set of integrated indexes, e.g. B-Tree index, bitmap index etc. However, for many applications this may not be sufficient as they may require a specialized "tailor-made" access to data to improve their performance significantly. Thus a possibility to implement and integrate new custom access method is needed.

1.1 Problem Outline

The design of extensible architectures represents an important area in database research. Relational database servers gained advanced functionality by introducing the object-relational data model with abstract data types. Thus, object-relational database systems can be naturally employed as platforms to design an integrated user-defined database solution.

As custom data types can be stored in relational tables along with the native ones, some applications may require a specialized index structures to be built on these data types to effectively handle frequent operations. As an example, we may consider a custom data type *polygon* representing a polygonal object in n-dimensional space, and custom predicate INTERSECTS which tests intersection of two objects of given type. Object-relational queries can be expressed in usual declarative fashion, e.g. "SELECT * FROM polygon_table p WHERE p.polygon_object INTERSECTS :*query_region*". Provided only with a functional implementation which evaluates the INTERSECTS predicate, the built-in optimizer of underlying database system has to include a full-table scan into the execution plan to perform given selection. In consequence, the resulting performance will be very poor for highly selective query regions.

Other needs for custom index types may arise when native data types are regarded in an application-specific manner, e.g. when table entries with standard data types are considered as points of a multidimensional space (this is equivalent to the case when range searches according to multiple attributes are frequent) or when an application carries out sophisticated access to text or LOB table items.

In order to achieve seamless integration of user-defined access methods, database systems provide developers with extensible indexing frameworks. An object-relational index type encapsulates stored functions for creating and dropping a custom index and for opening and closing index scans. Although the embedding of a custom index type is thus well supported, the actual implementation of its low-level functionality within a fully-fledged database kernel can be fairly complicated.

1.2 Implementation of New Access Method

When a new type of database index access method is needed for whatever reason, its actual implementation can be done according to three basic approach types. Particularly they are the integrating, the generic, and the relational approach as presented in [1]. This section discusses their advantages and disadvantages in relation to the difficulty of their implementation, the expected performance and their availability.

1.2.1 Integrating Approach

Outline: A new index access method is hard-wired directly into the kernel of an existing database system.

Implementation: Two types of implementation are distinguished: the *Extending Approach* and the *Enhancing Approach*. The enhancing approach is the easier one - many properties get inherited from an access method that already exists in the kernel; e.g. B-Trees can be enhanced to become a functional B-Trees. On the other hand the extending approach stands for real adding of a new access method which comprises sophisticated support for transactions (concurrency control, locking, recovery services etc).

Performance: The expected performance is the best possible in comparison with other approach types.

Availability: Code maintenance is a very complex task and requires access to low-level kernel components which is nearly impossible when the target database is not distributed as open source.

1.2.2 Generic Approach

Outline: To overcome the restrictions of the integrating approach, such called *Generalized Search Tree* (GiST) has been proposed as a generic way of implementation of a new index. GiST serves as a high-level framework to plug in block-based tree structures. It has to be built only once into a database kernel and already includes support for transactions.

Implementation: New index integration is quite easy; however the actual implementation of the GiST itself remains a very complex task.

Performance: Although the framework induces some overhead, GiST-based access methods can still be of high performance.

Availability: Due to its complex implementation, GiST exists only as a research prototype and it is an open question, if and when a comparable functionality will be a standard component of major commercial database systems.

1.2.3 Relational Approach

Outline: Custom index structure is mapped into a relational schema organized by built-in access methods and all the operations are done on top of a relational query language.

Implementation: No extensions to the database kernel are required and therefore an index can be implemented with less effort when comparing with other approach types.

Performance: The performance is questionable; however it should be sufficient as mentioned in [1].

Availability: By design, a relational access method is supported by any object-relational database system. It requires the same functionality as an ordinary database user or a relational database application.

1.3 Goals

In this work a new access method via the relational approach will be implemented. Particularly, the chosen index type is the UB-Tree which stands for a promising structure in the field of multidimensional access methods (MAMs). MAMs in common have high impact on different database application domains like data warehousing, data mining, or geographical information systems. However, they have not made their way into commercial database systems on a broad scale yet. The only exception is Transbase DBMS [2, 9] which comprises the native kernel implementation of just the UB-Tree.

All the advantages and drawbacks of relational index implementation will be studied in this work; then it will be compared with the native kernel integration, mainly with respect to performance issues.

CHAPTER 2

Relational Index

2.1 Relational Access Method

The basic idea of a relational access method is to delegate the management of persistent data to an underlying relational database system by implementing the index definition and manipulation on top of its SQL interface. In other words, an access method is called a *relational access method*, if any index-related data are exclusively stored in and retrieved from relational tables.

2.1.1 Basics

Relational access methods rely on the exploitation of the built-in functionality of existing database systems. Instead of extending any database kernel component, just the native data definition language (DDL) and data manipulation language (DML) with common object-relational enhancements in the sense of SQL:1999 (mostly the object types and collections) are employed to process updates and queries related to index data. This approach can be used for implementation of both basic services of all-purpose database systems and also very specialized application-specific extensions.

In other words, the SQL layer of the DBMS is used as a virtual machine for management of persistent data. It also means that a relational access method immediately benefits from any improvement of the underlying DBMS.

2.1.2 Relational Storage of Index Data

Relational access methods are designed to operate on relations rather than on dedicated disk blocks which is common to standard block-oriented access methods of a DBMS kernel. The actual persistent storage and block-oriented management of the relations are delegated to the underlying database server. The relational access method and the database system cooperate to maintain and retrieve the index data and all the functionality of the DBMS including concurrent transactions and recovery can be reused.

In order to support queries on index tables, a relational access method can employ any built-in secondary indexes, including hash indexes, B-trees, and bitmap indexes. Alternatively, payload data can be included into clustering by organizing index tables in a cluster or by storing them in index-organized tables.

2.1.3 Operations on Relational Access Method

According to previous specifications, a common block-oriented access method can be transformed to a relational access method by simple replacing each invocation of the underlying block manager by an SQL-based DML operation (e.g. calling of a function "blocks.get(*block_id*)" would be replaced by "SELECT * FROM blocks WHERE id = *:block_id*"). Thus the original procedural style of index operations remains unchanged, whilst all I/O requests are newly handled by the DBMS.

Such simple scenario however reduces the DBMS to a plain block manager and most of its functionality remains unexploited. To maximize the architecture-awareness, two types of declarative operations have been proposed in [1] in order to reduce the possible number of DML operations submitted from a procedural environment - particularly the cursor-bound and cursor-driven operations.

Cursor-bound operation stands for a query or an update related to a relational access method such that the corresponding I/O requests on the index data can be performed by submitting O(1) DML statements, i.e. by sequentially and concurrently opening constant number of cursors provided by the underlying DBMS. Its main advantages are:

- *Declarative semantics*. Operations are bound to the DML engine of the DBMS rather than to user-defined implementation code, therefore the DBMS gains responsibility for significant parts of the query processing. Thus the formal verification of the semantics is simplified if we can rely on the given implementation of SQL layer.
- *Query optimization*. Whereas the database engine optimizes the execution of single closed-form DML statements, a joint execution of multiple independently submitted queries is very difficult to achieve. By using only a constant number of cursors, the DBMS captures significant parts of the operational semantics at once.
- *Cursor Minimization*. The CPU cost of opening variable number of cursors or submitting several DML statements out of a stored procedure may become very high. For cursor-bound operations, the relatively high cost of opening and fetching multiple database cursors remains constant with respect to the complexity of the operation.

Cursor-driven operation is a special case of cursor-bound operation where the result can be retrieved as an immediate output of a *single* cursor provided by the DBMS. Particularly a query or an update related to a relational access method can be divided into two consecutive phases:

1. *Procedural phase*: In the first phase, index parameters are read, query specifications are retrieved and data structures required for the actual query execution may be prepared by user-defined procedures and functions. Additional DML operations on user data or index data are not permitted.
2. *Declarative phase*: In the second phase, only a single DML statement is submitted to the DBMS, yielding a cursor on the final results of the index scan which requires no post-processing by user-defined procedures or functions.

The major advantage of cursor-driven operations is their smart integration into larger execution plans. However, the ability to take advantages of cursor-driven operations heavily relies on the expressive power of the underlying SQL interface, often including availability of recursive queries.

2.2 Generic Schemes of Relational Index

In [1] two generic schemes for a relational storage of index data have been identified; particularly they are the navigational scheme and the direct scheme. This section discusses their main properties, advantages and disadvantages.

2.2.1 Navigational Scheme of Index Tables

Let $P = (T, R_1, \ldots, R_N)$ be a relational access method with a primary data table T and index related tables $R_1, \ldots, R_N$. P is called **navigational scheme** $\Leftrightarrow$ $(\exists\ t \subseteq T)$ $(\exists\ r_i \subseteq R_i, 1 \leq i \leq n)$: at least one $\rho \in r_i$ is associated with rows $\{\tau_1, \ldots, \tau_m\} \subseteq t$ and $m > 1$.

Therefore, a row in an index table of a navigational index may logically represent many objects stored in the primary table. This is typical in case of hierarchical structures that are mapped to a relational schema. In other words, an index table contains data that are recursively traversed at query time in order to determine the resulting rows. To implement a navigational query as a cursor-driven operation, a recursive version of SQL like SQL:1999 is required.

Although the navigational scheme offers a straightforward way to simulate any hierarchical index structure on top of a relational data model, it suffers from the fact that navigational data are locked like any other primary data. As two-phase locking on index tables is too restrictive, the possible level of concurrency is unnecessarily decreased. For example, uncommitted node splits in a hierarchical directory may lock entire sub-trees against concurrent updates.

A similar overhead exists with logging, as atomic actions on navigational data, e.g. node splits, are not required to be rolled back in order to keep the index tables consistent with the data table. Therefore, relational access methods implementing the navigational scheme are only well suited for read-only or single-user environments.

2.2.2 Direct Scheme of Index Tables

Let $P = (T, R_1, \ldots, R_N)$ be a relational access method with a primary data table T and index related tables $R_1, \ldots, R_N$. P is called **direct scheme** $\Leftrightarrow$ $(\forall\ t \subseteq T)$ $(\forall\ r_i \subseteq R_i, 1 \leq i \leq n)$: each $\rho \in r_i$ is associated with a single row $\tau \in t$.

It means that each row in the primary table is directly mapped to a set of rows in the index tables. Inversely, each row in an index table exclusively belongs to a single row in the primary table.

The drawbacks of the navigational scheme with respect to concurrency control and recovery are not shared by the direct scheme, as row-based locking and logging on the index tables can be performed on the granularity of single rows in the primary table. For example, an update of a single row *r* in the primary table requires only the synchronization of index rows exclusively assigned to *r*. As the acquired locks are restricted to *r* and its exclusive entries in the index tables, they do not unnecessarily block concurrent operations on other primary rows. In contrast to navigational indexes, the direct scheme inherits the high concurrency and efficient recovery of built-in tables and indexes.

CHAPTER 3

Multidimensional Indexing

3.1 Multidimensional Data

Idea of multidimensional indexing arises from the fact that data (records) can be considered as points of a multidimensional vector space. In terms of relational databases, each row of a database table can relate to a point of a multidimensional space, where each domain is represented by an attribute. Therefore such table stands for a subset of the space which is defined by Cartesian product of table columns.

3.1.1 Applications

Multidimensional approach can bring lots of benefits into various database applications. Often it is wise to consider and treat data as points of a multidimensional space. In some cases the mapping of data to a vector space is straightforward (geographical information systems, CAD databases etc.), in other cases the mapping is more or less synthetic but still useful (data warehousing, data mining, systems for information storage and retrieval, archives etc.). The common identifier for all such applications is that searching according to several criteria (i.e. according to more than one database attribute) is required quite often.

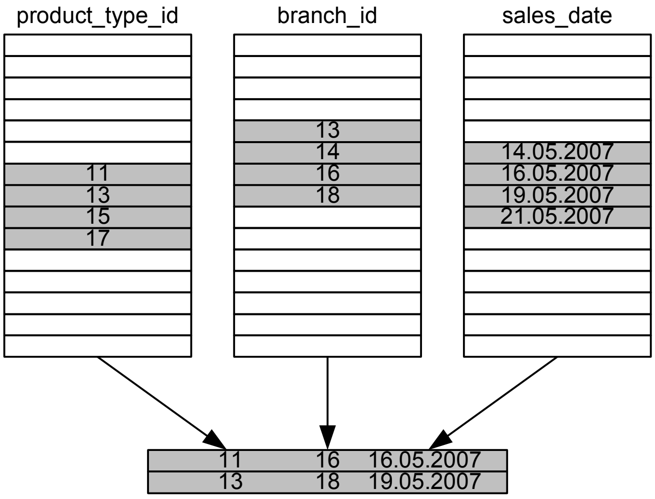
Let us consider following example: a database application that is used in a shopping company comprises table `sales` with attributes product_type_id, sales_date, branch_id (and possibly other ones). A common database query could be as follows:

```
SELECT * FROM SALES
WHERE PRODUCT_TYPE_ID BETWEEN 10 AND 20
      AND BRANCH_ID BETWEEN 15 AND 18
      AND SALES_DATE BETWEEN '14.05.2007' AND '21.05.2007'
```

It is likely that the result set of such query will be quite small in relation to the count of all entries in the `sales` table; in other words the *selectivity* of such query is small. This is the case when it would be wise to create an index on the columns of the table. The easiest solution available in all database systems is to create separate indexes on each attribute. An effective query plan would select temporary subsets of the table searched by particular attribute (with use of particular index) and then the final result would arise as an intersection of the temporary subsets.

Such principle of separate indexes is shown in Figure 1 (gray lines in upper tables correspond to search conditions according to particular attributes whilst the lower table stands for the result based on the intersection of all three attribute-related conditions).

Figure 1: Intersection of temporary result sets within query employing separate indexes



In real applications users usually want only few rows to be returned in the result set (they would hardly list thousands of entries to find the required ones). It means that even with growing count of attributes used in a search condition the expected output is still of approximately the same size. The main problem of above approach is that many rows from temporary result sets are often filtered out because they do not belong to the intersection.

Moreover, with more attributes in a search condition, the expected intersection is of smaller size and more and more rows are filtered out because they seldom fulfill all the conditions. When the count of entries filtered out becomes comparable with the count of all entries in a table then it is questionable whether the sequential passing of the whole table would not be faster.

3.1.2 Relational Databases

In multidimensional databases, objects are indexed according to several or many independent attributes. However, as mentioned in previous section, this task cannot be effectively handled by using many standalone indexes. Thus special indexing structures which would naturally index vectors of values instead of indexing single values have been required.

Common approach available in nearly all database systems is indexing of compound keys, i.e. several attributes are indexed by a single index - usually a compound B-Tree index (see chapter 3.2.2 "B-Tree with Compound Keys"). This method is more effective then utilization of separate indexes, however it still involves filtering of relatively high number of entries from temporary result sets.

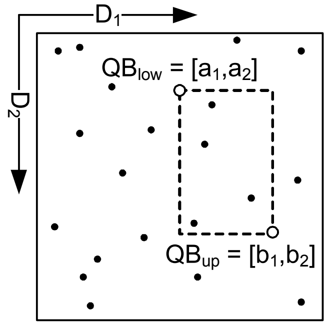
Therefore many access types has been introduced that are more suitable for indexing of several attributes at once; for example the KD-Tree, the R-Tree and its modifications, the UB-Tree etc. However, the integration of any such access method into the kernel of a commercial database system is a very costly task (see chapter 1.2 "Implementation of New Access Method"). Thus most of such advanced access methods are still in the state of research prototype or are available only as database plug-ins with restricted usability. Usually only a specific data types can be indexed (geometric objects of CAD or GIS databases), concurrency control and recovery services may not be presented at all. The only exception is the UB-Tree that was integrated into Transbase DBMS [2].

In this work, a cheaper way of developing new access method (particularly the UB-Tree via the relational approach) is investigated and compared with its native implementation.

3.1.3 Range Query

Range query (window query respectively) in a vector space is usually represented by a hyper-box in given space. The ranges of a query box *QB* are defined by two boundary points, the lower bound $QB_{\text{low}} = [a_1, a_2, \ldots, a_n]$ and the upper bound $QB_{\text{up}} = [b_1, b_2, \ldots, b_n]$ where $a_1 \leq b_1$, $a_2 \leq b_2$, …, $a_n \leq b_n$. The purpose of a range query is to return all points located inside the query box, i.e. to return all points *o* satisfying $a_i \leq o_i \leq b_i$, $i \in [1,n]$, as outlined in Figure 2.

Figure 2: Range query in 2-dimensional space



3.2 Common Index Methods

The most common method used for indexing of a single attribute is the B-Tree and its modifications. Its concept can be extended into the B-Tree with Compound Keys so that it is feasible to index more attributes at once. Basics of these approaches are described in this section.

3.2.1 B-Tree

B-Tree is a balanced search tree. Its internal nodes can have variable number of child nodes within some pre-defined range as mentioned later. The tree is balanced which means that all leaf nodes are at the same depth. Following rules have to be valid for proper B-Tree of degree *m*:

1 the root has at least 2 descendants unless it is a leaf

2 all inner nodes except from the root have at least $\lceil m/2 \rceil$ and at most *m* descendants

3 all branches are of the same length

4 all nodes except from the root have at least $\lceil m/2 - 1 \rceil$ and at most *m* data entries

5 data in a node are organized as p_0, (k_1,p_1,d_1), … (k_n,p_n,d_n) where:

- p_i is a pointer to a descendant
- k_i is a key (the keys are ordered in ascendant or descendant order)
- d_i stands for associated data
- (k_i,p_i,d_i) stands for data entry

6 let us consider $U(p_i)$ to be a sub-tree which is pointed to by p_i, then

- $\forall k \in U(p_{i-1})$: $k < k_i$
- $\forall k \in U(p_i)$: $k > k_i$

A modification to above principles is a B+ Tree. It differs in points (5) and (6) of B-Tree definition in following way:

5 data are stored in leaves only (or are referenced from leaves only), inner nodes comprise only keys and pointers

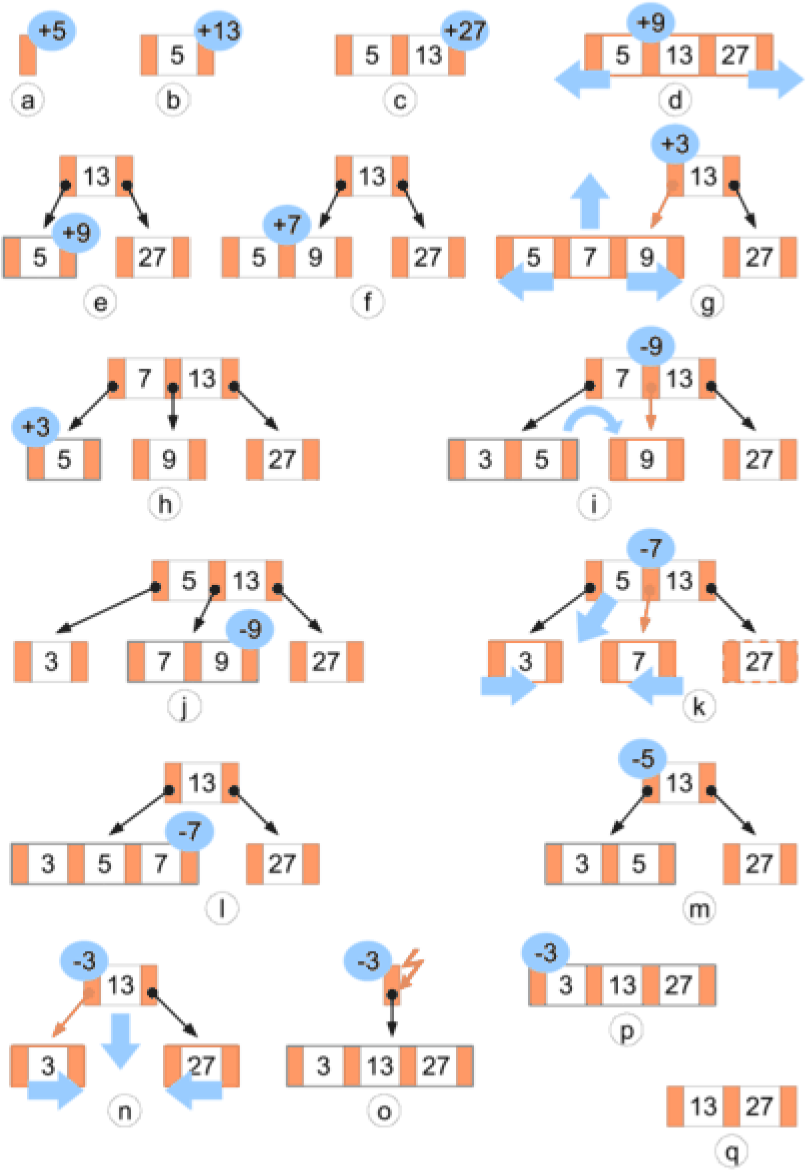
6 let us consider $U(p_i)$ to be a sub-tree which is pointed to by p_i, then

- $\forall k \in U(p_{i-1})$: $k \leq k_i$
- $\forall k \in U(p_i)$: $k > k_i$

Way of searching follows from above definitions and is performed in the typical manner, analogous to that in a binary search tree. Starting at the root, the tree is traversed top to bottom, choosing the child pointer whose separation values are on either side of the value that is being searched. The tree is traversed down to a leaf node (B+ Tree) or to a node containing searched data (B-Tree).

Figure 3 (adapted from [10]) shows operations of inserting and deleting entries.

Figure 3: Insertions and deletions in a B-Tree



3.2.2 B-Tree with Compound Keys

B-Tree presented in previous section does not allow indexing of multidimensional data. The easiest extension of B-Tree which enables this type of indexing is to consider its keys as a chained sequence of values of those attributes that are subject to index. When comparing two keys, they are compared linearly item by item.

Such extension is available in most database systems. Unfortunately, it brings some disadvantages. Probably the biggest one is an asymmetry in the order of the attributes. The first attribute is always the main one and serves for clustering of the vector space. It means that all data in the index are ordered only according to the first attribute, and not according to the other ones. Only in case when the first attribute (or all the previous attributes in general) has the same value for more entries, the index is sorted also according to following attribute for these entries. Therefore it is suggested to use an attribute with the smallest range of values at the first place so that the count of duplicities is as big as possible.

The asymmetry causes that in a range query many branches of the tree have to be searched through and this impacts the overall efficiency negatively. A solution would be to create several indexes, each for a different order of the attributes. However such approach is hardly affordable because of its enormous disk space demands.

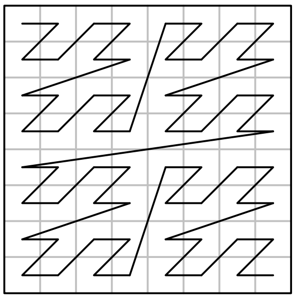
3.3 UB-Tree

UB-Tree [5, 4, 2] is one of the access methods that are natively used for indexing of multidimensional data - this section discusses its features and suitability for multidimensional queries.

3.3.1 Z-curve

The basic idea of the UB-tree is to use a space filling curve to map a multidimensional universe to one-dimensional space. Points of the universe are ordered according to such called Z-curve which preserves multidimensional clustering - it means that points that are close to each other in the original universe (using standard L_2-metric) are in general also close to each other on the Z-curve. Figure 4 shows the Z-curve for 2-dimensional universe of size 8×8.

Figure 4: Linear ordering of 2-dimensional space with a Z-curve



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

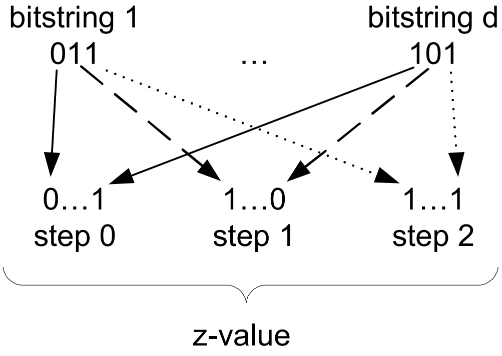
3.3.2 Z-value

Z-value (also called Z-address) is an ordinal number representing the position of a multidimensional point on the Z-curve. For proper determination of the Z-value, the universe has to be finite in each dimension. Let *d* be the count of dimensions of the universe and $x_i = x_{i,0} \ldots x_{i,s-1}$; $i \in [1,d]$ be the binary record of the value of a multidimensional point in dimension *i*. Then the Z-value can be counted according to following formula:

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

Above equation simply represents bit interleaving of values of the point in each dimension as shown in Figure 5 where *step* stands for bit position in given dimension (the most relevant bit position has its *step* equal to 0).

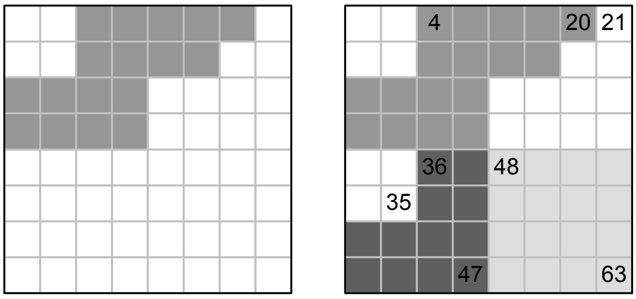
Figure 5: Computation of Z-value by bit interleaving



3.3.3 Z-region

Z-region [α : β] is a part of a multidimensional universe corresponding to all points of the universe within an interval on the Z-curve. The interval is defined by two boundary Z-values α and β (where α < β). The upper bound β is called *region address*. Set of Z-regions creates a disjunctive partitioning of the whole universe. Figure 6 shows Z-region [4 : 20] and partitioning of 2-dimensional universe into 5 Z-regions [0 : 3], [4 : 20], [21 : 35], [36 : 47] and [48 : 63].

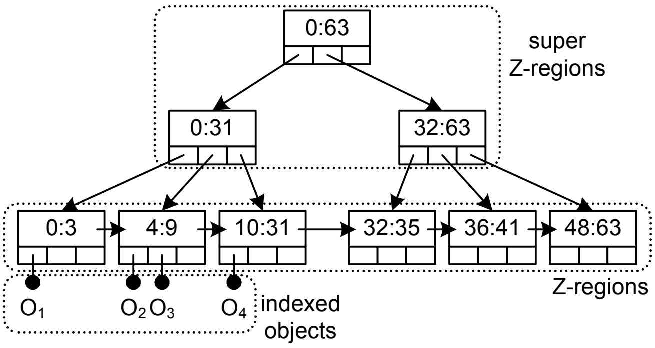
Figure 6: Z-regions in 2-dimensional space



3.3.4 Tree Structure

The structure of UB-Tree is similar to the standard B-Tree (B+ Tree modification actually). Leafs of UB-Tree represent the Z-regions containing indexed objects, whilst inner nodes of UB-Tree represent such called *super Z-regions*. A *super Z-region* comprises all (super) Z-regions that lie entirely inside it. Therefore the UB-Tree structure is determined by a nested Z-region hierarchy as shown in Figure 7.

Figure 7: Hierarchical UB-Tree structure



Algorithms that handle insertions, deletions and updates are the same as for B-Trees - the only difference is that at first the Z-value is computed based on indexed attributes of a database entry and then such Z-value is used as a key in subsequent "B-Tree operation".

3.3.5 Formal Definition

Since the UB-Tree stands for a smart extension of the B+ Tree to handle multidimensional objects, their formal definitions are quite similar. Particularly, the first 5 points of B+ Tree definition (see chapter 3.2.1 "B-Tree") are shared by the UB-Tree, just the last one is different:

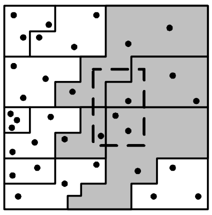
6 let us consider $U(p_i)$ to be a sub-tree which is pointed to by p_i, and $Z(d)$ to be a function that computes Z-value for a multidimensional point d, then

- $\forall k \in U(p_{i-1})$: $k \leq k_i$ & $\forall d \in U(p_{i-1})$: $Z(d) \leq k_i$
- $\forall k \in U(p_i)$: $k > k_i$ & $\forall d \in U(p_i)$: $Z(d) > k_i$

3.3.6 Range Query

Unlike the operations of insert, update and delete, a range query cannot be simply forwarded to the B-Tree. As mentioned previously, the range query in a multidimensional space is defined by two boundary points $QB_{low} = [a_1, a_2, \ldots, a_n]$ and $QB_{up} = [b_1, b_2, \ldots, b_n]$ and its purpose is to return all points lying inside such query box. In terms of UB-Tree a range query can also be defined as a search through all UB-Tree Z-regions that intersect given query box as shown in Figure 8.

Figure 8: Z-regions intersecting a query box in 2-dimensional space



Following algorithm was presented by Markl in [2]: let us consider *ql* and *qb* to be the Z-addresses related to QB_{low} or QB_{up} respectively. At first the Z-region containing *ql* is located (note that *ql* does not need to relate to a point existing in the database - we only need to know which Z-region it belongs in). This Z-region is searched for relevant objects that really lie inside the query box, the others are filtered out. Then a subsequent Z-region intersecting the query box is retrieved and processed etc. The algorithm iterates until the upper bound of currently processed Z-region is higher than *qh*.

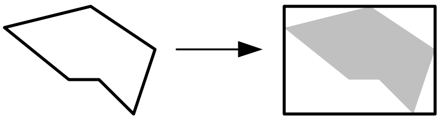
The crucial part is the way of obtaining a subsequent Z-region intersecting the query box. In Markl's work this is done by calculation of the Z-value for next intersection point of the Z-curve with the query box based on the currently processed Z-region; then a Z-region containing the computed Z-value is obtained.

Similar approach has been employed in this work as well. Please refer to the chapter describing the UB-Tree index implementation (see chapter 4 "Implementing the Relational UB-Tree Index") for more details on both the above algorithm and the function used for calculation of next intersection point *get_next_zvalue* (see chapter 4.5.1 "Get Next Z-value").

3.3.7 Processing Multidimensional Objects

In database area, indexing of multidimensional objects is often simplified to indexing of their minimum bounding boxes (MBB). A MBB is the smallest cube (in a multidimensional space) which completely covers the original object, usually with sides parallel to the axis. This approach reduces demanding computation of objects intersection, position etc. An example of MBB in 2-dimensional space can be seen in Figure 9.

Figure 9: Minimum bounding box in 2-dimensional space

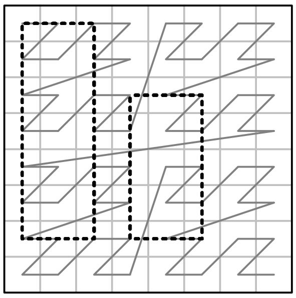


Unfortunately, even indexing of MBB in the UB-Tree brings problems. When a UB-Tree page (i.e. a Z-region) has to be split because the count of items inside it exceeds the maximal allowed count, we may face a problem of choosing the divider for splitting algorithm.

Let us imagine that two objects have been identified so that the dividing point should be found on the Z-curve between these objects. The trouble is that the Z-curve may go forth and back from one object to the other one and therefore such divider cannot be found so that one object lies entirely on its "left side" and the other object on its "right side". Thus at least one of the objects would belong to both Z-regions that originate from the splitting of the original Z-region and we may identify situations where even many objects would belong to both Z-regions. Then the splitting algorithm would not work as expected and high redundancy would be involved in such UB-Tree.

Similar problem arises when one MBB lies inside another MBB or when two MBB intersect each other. A simple example of two problematic objects is shown in Figure 10.

Figure 10: Intersection of Z-curve with two minimum bounding boxes



A possible solution is to consider a MBB with boundaries $QB_{\text{low}} = [a_1, a_2, \ldots, a_n]$ and $QB_{\text{up}} = [b_1, b_2, \ldots, b_n]$ to be a point of higher dimension with coordinates $[a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n]$. The range query mentioned in previous section has to be modified in this case as well. Thorough information about this approach can be found in [3].

CHAPTER 4

Implementing the Relational UB-Tree Index

4.1 Oracle Database Platform

Relational UB-Tree index was implemented with Oracle database platform [8] as the underlying DBMS (particularly the free version Oracle 10g XE has been employed in this work). It has been chosen for three main reasons:

- it natively supports object extensions via its PL/SQL language
- it supports recursive SQL queries which are required for optimal implementation of navigational type of a relational index
- SQL query plans, processing etc. can be influenced by a programmer with a set of integrated tools (e.g. optimizer hints)

PL/SQL (Procedural Language/SQL) stands for proprietary server-based procedural extension to the SQL database language. Its syntax strongly resembles that of Ada and supports variables, arrays, conditions, loops and exceptions. It also includes features associated with object-orientation. More details about the syntax and usage of PL/SQL can be found at [11].

Recursive SQL queries offer a way to traverse tree-like structures with one SQL statement. In Oracle they have proprietary syntax which differs from the SQL: 1999 standard. It comprises two clauses used for definition of recursive traversing, particularly START WITH <condition> and CONNECT BY <condition>. In general, evaluation of such query is done in following way:

1. Oracle selects the root row(s) of the hierarchy - i.e. those rows that satisfy the condition of the START WITH clause.
2. Oracle selects the child rows of each root row. Each child row must satisfy the condition of the CONNECT BY clause with respect to one of the root rows.
3. Oracle selects successive generations of child rows. At first it selects the children of the rows returned in step 2, and then the children of those children, and so on.
4. If the query contains a WHERE clause, Oracle eliminates all rows from the hierarchy that do not satisfy the condition of the WHERE clause.

The syntax of a recursive statement is as follows:

```
SELECT <what> FROM <table>
WHERE <filter>
START WITH <condition>
CONNECT BY <condition>
```

4.2 Common Properties of UB-Tree Index

In this work several approaches to a relational UB-Tree index implementation are presented. This section describes basic features that are common for all chosen methods, whilst particular implementation details are mentioned in subsequent sections.

4.2.1 Multidimensional Tuple

Multidimensional tuple is the base element in probably all applications designed to handle and store multidimensional data. It represents a logical point in a multidimensional space which should be involved in multidimensional indexing. The tuple is compound of items representing the value of the point in corresponding dimensions of the space.

Concerning the UB-Tree, type of each item of a tuple has to be numeric or any type that can be converted to numeric type (e.g. date, enumeration, string of fixed length and fixed number of allowed characters etc.). Moreover, the range of values in each dimension (i.e. the minimal and the maximal value of an item) has to be known in advance. These restrictions arise from the characteristic of the Z-curve - it is necessary to know both where the Z-curve begins (the 0 point) and where it ends (actually the length of the maximal Z-value has to be known in some algorithms mentioned later, however these restrictions are identical); otherwise it would not be possible to assign appropriate Z-value to a tuple.

Without loss of generality only the numeric values for tuple items are considered and ranges of values of items in all dimensions are supposed to have the same *length* - it equals to the nearest higher exponent of 2 of maximal range of a domain, i.e.:

$$\textit{length} = \min(\{2^{exp} \mid exp \in \mathrm{N},\ 2^{exp} \geq \max_{dim}\text{-}\min_{dim}\ \forall\ dim\})$$

4.2.2 Defining the Constraints

As mentioned in previous section, restrictions for the range of the value have to be defined in all dimensions of a multidimensional tuple. However, this task cannot be achieved by simple definition of some standard database constraints on primary data table as if it could be done in case the UB-Tree index was implemented directly into the database kernel similarly to [2].

Therefore a separate table has to be created to hold the restrictions. This table consists of just 3 columns - an identifier of a dimension, its lower bound and its upper bound:

```
CREATE TABLE ub_constraints (
      dimension_id NUMBER,
      lower_bound NUMBER,
      upper_bound NUMBER
);
```

4.2.3 Integrating the Index with a Database

A multidimensional tuple serves as an interface between tables storing the primary data and tables storing the index data. It means that if a row is inserted into (or updated in) the primary table, then a tuple representing row data that are subject to multidimensional index should be generated and index data based on the value of such tuple along with an identifier of primary data row should be inserted into (or updated in) the index table. This task can be easily achieved by defining appropriate AFTER TRIGGERs on the primary table. The only restriction is that the identifier of a row in the primary table is supposed to be just one NUMBER; in case the PRIMARY KEY of the primary table consists of more attributes or is of a different type, an alternative numeric identifier for a primary data row has to be created. Then it is necessary to define required constraints via the ub_constraints table mentioned in previous section. An example can be seen in Appendix A.

Another way of integrating the index into a database would be exploiting of an extensible indexing architecture of given DBMS. Currently many commercial database systems provide an interface which enables developers to register custom secondary access methods, however the effort of implementing such type of index is behind the scope of this work because the goal is mainly to compare a relational index with its native implementation - a custom index type implementation via the extensible framework would not bring any advantages in comparison with usage of the AFTER TRIGGERs.

In this work several, ways of relational UB-Tree implementation are presented and for higher transparency each of them serves as a *black box* for index data maintenance. Therefore a common interface is defined for all approaches. It provides a set of functions to keep index data consistent with primary data whenever the primary data are changed, and also a function to obtain identifiers of rows in primary table based on the query specification (the actual usage of the index in a SELECT statement).

Particularly the functions for index data maintenance which should be used in AFTER TRIGGERs are *insert_tuple*(*tuple*, *id*), *update_tuple*(*tuple*, *id*) and *delete_tuple*(*id*), whilst the function *inside_query_box*(*lower_bound*, *upper_bound*) serves for obtaining identifiers of primary data rows based on a query box determined by its lower and upper boundaries (which both are actually multidimensional tuples). The index is then supposed to be used in following way:

```
SELECT primary.*
FROM TABLE(inside_query_box(Type_tuple(X_1,X_2,…,X_n),
      Type_tuple(Y_1,Y_2,…,Y_n))) index
LEFT JOIN PRIMARY_TABLE primary
      ON index.id = primary.id
```

More about the real processing of above concept can be found in subsequent sections describing the particular UB-Tree index implementations. Short specification of user data types is presented in Appendix B.

4.3 UB-Tree via the Direct Scheme

One of the ways proposed in [1] to implement a relational index is the direct scheme (see chapter 2.2.2 "Direct Scheme of Index Tables") where data of the index are exclusively related to primary data and do not form any specific structure. Description of this approach for developing the UB-Tree index can be found in this section.

4.3.1 Basic Concept

According to the definition of the *direct scheme*, each row in the index table is associated with only one row in the primary table. In this implementation of the relational UB-Tree, the mapping of primary data into index data is moreover bijective and quite simple. Only the actual Z-value of a tuple is computed and this Z-value is stored to the index table along with the identifier of the primary row.

The idea of obtaining identifiers of primary data lying inside a query box is also straightforward - the query box is partitioned into several continuous Z-regions and the index table is queried to get all rows which Z-value lie between the boundaries of such Z-regions.

4.3.2 Index Table

As outlined in previous section, the index table consists of just 2 columns - the Z-value and the identifier of a primary data row:

```
CREATE TABLE ub_index (
      z_value Type_z_value,
      primary_id NUMBER
);
```

The PRIMARY KEY constraint should be defined on the *z_value* attribute to exploit the power of the underlying database engine when performing a SELECT query with multiple "WHERE *z_value* BETWEEN" conditions based on the partitioning of the query box. For the ease of implementation, if we do not want to implement a custom index type for our Type_z_value data type, a functional index can be defined instead of the primary key on the column *z_value* where the function simply transforms the *z_value* into a string; consequently, all concerned query conditions should be transformed to be in form "WHERE *z_value.to_string*() BETWEEN" so that the functional index could be exploited.

The *primary_id* attribute could be defined as FOREIGN KEY to the PRIMARY KEY of the primary table and an ON DELETE CASCADE constraint could be used instead of AFTER DELETE trigger for deleting a row from the index table when a row in the primary table is deleted, however this approach is not in compliance with the *black box* concept and strict separating of primary and index data.

If this type of UB-Tree index is used in an environment where changes and deletions of rows in the primary table are frequent, it may be wise to define a standard secondary database index on the *primary_id* attribute, because the `ub_index` table is queried according to this attribute when an INSERT or UPDATE of the UB-Tree index structure is triggered. On the other hand this is not recommended in environments with majority of insertions because such secondary index could become very large and the insertions would be slower just because of updates of this index.

4.3.3 Inserting, Updating and Deleting a Tuple

According to the main principle of this approach, all these operations are quite simple:

- in case of inserting or updating, the actual Z-value of a tuple being processed is computed and is stored to the index table
- in case of deleting only the row with the corresponding identifier is deleted from the index table

4.3.4 Querying Tuples

The main idea is to partition the query box into several continuous Z-regions. Then the index table is queried to get all rows which Z-value lies between the boundaries of such Z-regions. The actual query statement could be as follows:

```
SELECT primary_id
FROM ub_index u,
      TABLE(decompose_query_box(:lower_bound, :upper_bound)) d
WHERE u.z_value BETWEEN d.lower AND d.upper
```

The function *decompose_query_box* generates a temporary table that contains rows with lower and upper bound specification of particular Z-regions covering the query box which is defined by 2 multidimensional tuples representing its *lower_bound* and *upper_bound* boundaries.

The easiest way to decompose a query box is repeated calling of functions g*et_next_zvalue* (see chapter 4.5.1 "Get Next Z-value") and *get_next_zvalue_out* (see chapter 4.5.2 "Get Next Z-value Out"). This approach guarantees that the query box is covered by sequence of optimal Z-regions, which means that:

- each of such Z-regions is as long as possible
- it does not exceed the query box in any dimension

However, the experiments have shown that this is also the least efficient way. Even though both of the functions are of linear time complexity in relation to the bit length of the maximal Z-value, the characteristics of the Z-curve cause that a query box is decomposed into huge number of Z-regions and thus the whole computation and query evaluation take a lot of time.

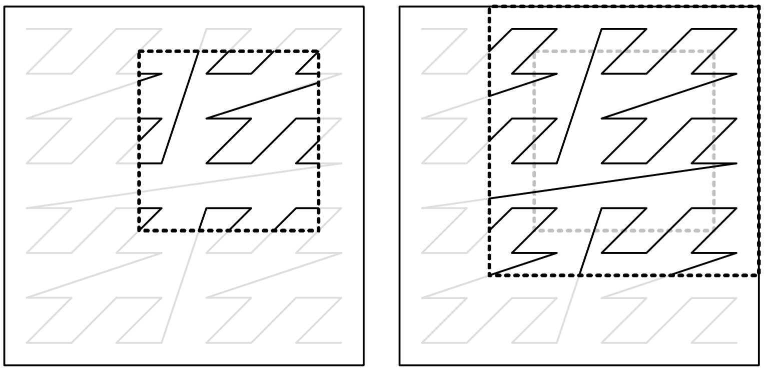
For example a query box in 5-dimensional space with range of values equal to 10 in each dimension has been divided into approximately 25.000 Z-regions. It means that the average length of a Z-region has been about 4. Although this simple experiment cannot impact the actual result in general case (the query box can be covered just by few Z-regions even though it is much larger), we may estimate that this is not the right way.

The main problem in above scenario is that the average length of a Z-region is too short. Therefore it would be wise to define minimal allowed length of a Z-region providing that the Z-region may exceed the boundaries of the query box in some dimensions.

Whole space could be divided by a grid into multidimensional cubes with size equal to 2^{const} in each dimension, where *const* is an integer higher or equal to 1. Each such cube is covered by just 1 continuous Z-region which length is considered to be the minimal allowed one. It means that the actual minimal length equals to $(2^{const})^{dim}$, where *dim* is the count of dimensions of the space. Moreover, if we define total ordering of the cubes according to the Z-curve and a query box has non-empty intersection with several subsequent cubes in relation to such ordering, then the related Z-region covering a part of the query box has its length equal to the sum of lengths of the subsequent cubes. Such Z-regions generate an *extended query box*.

If this logic is applied to the same query box as in the example mentioned in one of the previous paragraphs, and in case the *const* equals to 3, then the box is completely covered just by 1 Z-region. An example is shown in Figure 11.

Figure 11: Query box partitioned into 9 Z-regions and its extended query box partitioned into 4 Z-regions when const=1



Above approach requires post-filtering of selected tuples to ensure that they really belong to the original query box. Therefore an optimal *const* should be found so that both the number of Z-regions covering the extended query box is rather small and the number of tuples discarded because of post-filtering is not significant in relation to the number of all tuples lying inside the extended query box. A discussion about choosing the proper *const* can be found in Experiments section of this work (see chapter 5.2.1 "Optimal Constant for Extended Query Box in Direct Scheme").

The actual algorithm for *decompose_query_box* function utilizes the method *"Divide and conquer"* where the whole space is divided into 2 multidimensional sub-cubes of the same size and each sub-cube is then processed recursively. When a sub-cube with minimal allowed size is being processed and in case it has non-empty intersection with the query box, then its boundaries are sent to the output.

For higher effectiveness both positive and negative pruning are employed in this algorithm. It means that if the cube being currently processed does not intersect the query box, then the further processing of such cube is skipped. On the other hand if the cube is nearly whole covered by a part of the query box (it means that all minimal sub-cubes of such cube intersect the query box), then it is whole sent to the output.

Simplified pseudo code of this algorithm which does not consider optimization of the query box intersection with subsequent sub-cubes is as follows:

```
function decompose_query_box(lower_bound, upper_bound) {
      function decompose(query_box, cube) {
            if (cube does not intersect query_box)
                  return;
            if (cube is minimal or cube is covered by query_box) {
                  send cube boundaries to the output;
                  return;
            }

            decompose(query_box, lower-sub-cube);
            decompose(query_box, upper-sub-cube);
      }
      query_box = BOX(lower_bound, upper_bound);
      cube = BOX(minimal_z_value, maximal_z_value);
      decompose(query_box, cube);
}
```

For estimation of time complexity of above algorithm following definitions are needed:

- The *basic cube* is a multidimensional cube with size equal to 2^{int} in each dimension, where *int* is an integer equal to or higher than 0; moreover such cube has to be filled by just one continuous Z-region. The length of such Z-region can be easily counted and equals to $(2^{int})^{dim}$.
- Let *cnt* be the count of maximal *basic cubes* which Z-regions completely cover the extended query box. From the construction of the extended query box follows that the smallest possible size of a *basic cube* in the extended query box is 2^{const} and the length of related Z-region is $(2^{const})^{dim}$.
- Let *n* be the bit length of maximal Z-value.

The time complexity of each execution of *decompose* function is O(*n*) because:

- The condition "cube does not intersect query_box" is counted as the result of the function *get_next_zvalue* (see chapter 4.5.1 "Get Next Z-value") which has complexity O(*n*).
- The condition "cube is minimal" is of constant complexity because just an integer representing the minimal length is added to a Z-value (constant complexity) and then two Z-values are compared (again constant complexity).

- Finally, the "cube is covered by query_box" condition is again of O(n) complexity because it utilizes the function *inside_box* (see chapter 4.5.3 "Inside Box") executed on both the boundaries of the cube.

Because of both positive and negative pruning the recursive dividing of the cube is executed at most *cnt*-times in each level of depth of nested calling of the function, in other cases one of the conditions evaluates to true and the function ends. The maximal depth of nested calling can be also counted and equals to $n/const$ however the *const* factor can be omitted in this computation.

Therefore the overall time complexity of *decompose_query_box* function is $O(cnt*n*n) = O(cnt*n^2)$ which means that the time complexity is quadratically dependent on the length of maximal Z-value, and linearly dependent on the complexity of the extended query box.

4.4 UB-Tree via the Navigational Scheme

Another way proposed in [1] to implement a relational index is the navigational scheme (see chapter 2.2.1 "Navigational Scheme of Index Tables") where data of the index create a hierarchical structure. Description of this approach for developing the UB-Tree index can be found in this section.

4.4.1 Basic Concept

According to the definition of the *navigational scheme*, each row in the index table is associated with one or more rows in the primary table. The chosen concept is similar to the implementation of the relational R-tree in [1].

The rows in the index table form a hierarchical structure that stands for the natural way of implementing the UB-Tree which is essentially also hierarchical. Each row contains logic identifier of the page of UB-Tree, its level in the tree, reference to the page that stand for its direct descendant and boundaries of related Z-region. On the lowest level the reference actually contains identifier of a row in the primary table, the lower boundary contains the Z-value of indexed data whilst the upper boundary is null.

There are three ways of obtaining identifiers of primary data lying inside a query box - either a recursive SQL query statement can be used, or the UB-tree structure can be traversed programmatically or a database cursor can be opened to obtain required data.

4.4.2 Index Table

The index table for the navigational UB-Tree is designed in following way:

```
CREATE TABLE ub_index (
      page_id NUMBER,
      level NUMBER,
      son_id NUMBER,
      z_lower Type_z_value,
      z_upper Type_z_value
);
```

The *page_id* attribute stands for the identifier of a logical UB-Tree page. Let *max* be the maximal count of items in one logical UB-Tree page. Then there can be several rows in the table with equal *page_id* up to the value of *max*. This concept has been chosen because it allows flexible assignment of the page size and the query to obtain searched objects can be written in a smart way, as mentioned later.

Another possibility is to transform the schema into non-first normal form and have just *page_id* as the PRIMARY KEY. However several boundary sets consisting of *son_id*, *z_lower* and *z_upper* would have to be stored along with each row and thus the change of maximal items count in UB-Tree page would be quite difficult because it involves changes in the code. Moreover queries on this table would be less transparent as the WHERE clause would comprise several conditions related to each boundary set.

To minimize disk access cost during most SQL queries, clustering of the table is defined according to *page_id* attribute. This approach ensures that entries belonging to one logical UB-Tree page are stored in one physical cluster on the disk.

The PRIMARY KEY constraint is composite and comprises the attributes *page_id* and *son_id*. Another INDEX is defined on the *son_id* attribute because the updates of the UB-Tree structure related to INSERT, UPDATE or DELETE of a row in the primary table involve number of queries according to this attribute.

4.4.3 Inserting a Tuple

At first a logical UB-Tree page has to be found where the z-value of a tuple being inserted belongs. This can be easily achieved with recursive SQL query:

```
SELECT * FROM ub_index
WHERE level = 1
START WITH page_id = 1
CONNECT BY
      PRIOR son_id = page_id
      AND :tuple_z-value BETWEEN z_lower AND z_upper
```

If the count of items in a logical page exceeds *max*, then the page has to be split into two pages and a new divider has to be inserted into the parent page. This may involve recursive splitting of the ancestor pages up to the root. The algorithm is similar to the algorithm of splitting standard B-tree page. The interesting part is the way of choosing the divider. Similar algorithm to the one proposed in [2] has been used - a divider that causes the least possible partitioning of the space is chosen. If we consider the definition of the *basic cube* mentioned in one of the previous chapters (see chapter 4.3.4 "Querying Tuples"), then the Z-region covering the original page is divided into two Z-regions that cover maximal possible *basic cubes*.

4.4.4 Deleting a Tuple

The tuple is simply deleted from the index table according to its identifier.

```
DELETE FROM ub_index
WHERE son_id = id AND page_lev = 0
```

If the count of items in the concerned logical UB-Tree page is lower than *max*/2, then some items has to be transferred to the page from a neighbor page, provided that the count of items in the neighbor page is sufficient; otherwise these two pages have to be merged and their divider has to be removed from the parent page. This may involve recursive merging of the ancestor pages up to the root. The algorithm is similar to the algorithm of merging standard B-tree pages and does not include any special solution.

4.4.5 Updating a Tuple

This algorithm comprises subsequent calling of delete tuple and insert tuple functions and therefore it does not need to be described more thoroughly.

4.4.6 Querying Tuples via Recursive SQL Statement

Probably the most interesting part of this approach is the way of obtaining identifiers of tuples lying inside a specified query box. As mentioned in both [1] and [2] the crucial problem of many relational index solutions is the number of context switches between user defined functions and the database kernel if the index tables are traversed programmatically. Therefore just one recursive SQL query statement has been created that utilizes the functions *get_next_zvalue* (see chapter 4.5.1 "Get Next Z-value") and *inside_box* (see chapter 4.5.3 "Inside Box"). Particularly the statement is as follows:

```
SELECT son_id FROM ub_index
WHERE level = 0
START WITH page_id = 1
CONNECT BY
      PRIOR son_id = page_id AND
            ((get_next_zvalue(z_lower, :lower_bound,
                  :upper_bound, 0) <= z_upper
                  AND PRIOR level > 1)
            OR
            (inside_box(z_lower, :lower_bound, :upper_bound) = 1
                  AND PRIOR level = 1))
```

The first part of the OR condition within the CONNECT BY clause serves for actual descending the UB-Tree hierarchy. Even though it may not be seen at first sight, the Z-region corresponding to a page is simply checked on intersection with the query box via the *get_next_zvalue* function executed on its lower bound and subsequent comparison of the result with its upper bound. The fourth optional parameter (the 0) in the calling of *get_next_zvalue* means that the returned Z-value can be equal or higher to the actual Z-value of *z_lower* attribute; in original algorithm proposed in [2] only higher Z-values are considered.

The second part of the OR condition filters out the tuples from the lowest level of the UB-Tree that belong to the page but do not lie inside the query box (the filtering is done via the *inside_box* function).

4.4.7 Querying Tuples in Procedural Way

As mentioned before, the procedural traversing of the index structure is not recommended in both [1] and [2] because of high number of context switches. However many programmers would probably consider only this way and therefore it is tested in this work as well.

The traversing is similar to the recursive SQL processing in previous case - algorithm starts with the root page and continues with all logical descendant pages such that their Z-region intersects the query box until a page on the lowest level is reached; all tuples from such page are then tested whether they really belong to the query box.

Moreover a positive pruning takes place in the algorithm - if the Z-region of a page on any level of the UB-Tree hierarchy lies completely inside the query box then all tuples belonging to its descendant pages on the lowest level are sent directly to the output.

These techniques are employed in the algorithm:

- The test for intersection of a Z-region with the query box is done via the function *get_next_zvalue* (see chapter 4.5.1 "Get Next Z-value") as in previous case.
- The test whether a Z-region lies completely inside the query box is done via the function *get_next_zvalue_out* (see chapter 4.5.2 "Get Next Z-value Out") - the function is executed on the lower bound of a Z-region and then the result is compared with the upper bound of the same Z-region.
- All tuples from descendant pages to a page which Z-region is inside the box are obtained by simple SQL range query. An index should be defined on *z_lower* attribute to make this range query effective; for the ease of implementation just the functional index which converts Z-value to a string was used in the relational UB-Tree.
- Filtering of tuples on the lowest level is done via the function *inside_box* (see chapter 4.5.3 "Inside Box") similarly to previous case.

The main algorithm is then following:

```
function inside_query_box(lower_bound, upper_bound) {
      function process_page(page) {
            if (page.level == 0)
                  "filter and output all tuple IDs from page";
            else for (all sub-pages of the page) {
                  if (sub-page completely inside query box)
                        "output all tuple IDs which z_value lies
between boundaries of the sub-page; the IDs are obtained from the
range scan on whole ub_index table";
                  else if (sub-page intersects query box)
                        process_page(sub-page);
            }
      }
      process_page(root);
}
```

4.4.8 Querying Tuples via a Database Cursor

Because the positive pruning mentioned in previous section cannot be simply employed in the recursive SQL query, another way that can be used for obtaining the tuples lying inside a query box is a combination of the recursive query and programmatic traversing of the UB-Tree. An implicit database cursor is opened on the `ub_index` table and then either tuple filtering is executed on the lowest page of the UB-Tree or a range scan based on boundaries of processed Z-region is performed.

Simplified pseudo code follows:

```
for (row in
      SELECT * FROM ub_index
      WHERE level = 0
      // positive pruning in main condition:
      OR (level > 0
            AND z_lower.inside_box(:lower_bound, :upper_bound) = 1
            AND z_lower.get_next_zvalue_out(:lower_bound,
                  :upper_bound) >= z_upper)

      START WITH page_id = 1
      CONNECT BY
            PRIOR son_id = page_id AND
                  ((get_next_zvalue(z_lower, :lower_bound,
                        :upper_bound, 0) <= z_upper
                        AND PRIOR level > 1

                        // positive pruning in connect by condition:
                        AND (PRIOR z_lower.inside_box(:lower_bound,
                              :upper_bound) = 0
                              OR PRIOR z_lower.get_next_zvalue_out
                                    (:lower_bound, :upper_bound)
                                    < PRIOR z_upper))
                  OR
                  (inside_box(z_lower, :lower_bound,
                        :upper_bound) = 1
                        AND PRIOR level = 1)))
{
      if (row.level == 0)
            "output tuple ID associated with the row";
      else
            // positive pruning
            "output all tuple IDs which z_value lies between
boundaries of the Z-region associated with the row; the IDs are
obtained from the range scan on whole ub_index table";
}
```

As we may see, the condition for positive pruning in the cursor operation has to be presented:

- in the WHERE condition of the SQL statement, so that the logical UB-Tree page fulfilling given criteria is output from the cursor;
- in the CONNECT BY condition of the SQL statement (the condition is presented in the negative sense), so that the recursive query is traversed further only in case when the pruning condition is not fulfilled;
- in the actual code processing the cursor output (the condition is simplified to test the *level* of a row that got into the output), so that we know whether the range query should be evaluated.

Similarly to procedural traversing of the UB-Tree structure, the pruning condition comprises the function *get_next_zvalue_out* (see chapter 4.5.2 "Get Next Z-value Out"), which is executed on the lower bound of a Z-region and then the result is compared with its upper bound; and also the function *inside_box* (see chapter 4.5.3 "Inside Box"), which is executed just on the lower bound.

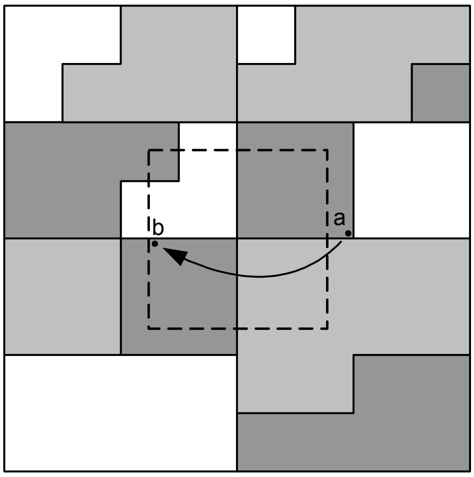
4.5 Algorithms for Processing Z-value

During the UB-Tree index implementation several algorithms that handle Z-value have been used. The most important ones are described thoroughly in this section.

4.5.1 Get Next Z-value

Function *get_next_zvalue*(*z_value*, *lower_bound*, *upper_bound*) generates the nearest higher Z-value (to the specified *z_value*) that lies inside a query box determined by its *lower_bound* and *upper_bound* boundaries; in other words it calculates Z-value *nip* of the next intersection point with the query box (see Figure 12). The function was presented in [2] however the description was rather vague and contained mistakes. Because it is an essential part of all mentioned UB-Tree range query algorithms, it deserves a deeper insight in this work.

Figure 12: Example of starting point (a) and searched point (b) in get_next_zvalue function



For detailed description of the algorithm several functions have to be introduced:

- Function *bit_position*(*dim*, *step*) returns the bit position in Z-value that corresponds to a dimension *dim* and a *step* in a tuple item from the dimension *dim*. Value of *step* is derived from the construction of Z-value: it refers to a bit position in a tuple item where the most relevant bit has its *step* equal to 0; for more details refer to definition of the Z-value (see chapter 3.3.2 "Z-value").
- Conversely, given a bit position *bp* in a Z-value, the functions *get_dimension*(*bp*) and *get_step*(*bp*) return the corresponding dimension, respectively step.
- Let us consider {} to be an operator that returns value of a number on specified bit position; e.g. *z_value*{*bp*} returns the bit value of *z_value* on *bp* bit position.

Now the algorithm itself can be described:

- At first the original *z_value* has to be incremented by one, i.e. *nip* = *z_value* + 1 (this step is omitted in some special cases as mentioned previously).

- Then *nip* is tested whether it already lies inside the query box by bitwise comparing with the Z-values of *lower_bound* and *upper_bound*.

  The main idea is quite simple and *nip* does not need to be transformed back to Cartesian coordinates. Moreover additional information are determined during the computation. For each dimension several properties are held and they are being updated whilst *nip* is being processed bit by bit from the most relevant bit to the least relevant one according to the comparison of value of *nip* on actual bit position with the value of *lower_bound* (or *upper_bound* respectively) on the same bit position.

  The attributes that are being distinguished for each dimension *dim* are following:

  - `flag`[*dim*] indicates the actual position of *nip* in a dimension *dim* - the value is 0 if *nip* is inside the query box in dimension *dim*; the value is -1 if *nip* has fallen below the minimum of the query box in dimension *dim*; the value is 1 if *nip* has exceeded the maximum of the query box in dimension *dim*; initially the value is set to 0 for each dimension
  - `out_step`[*dim*] holds the step in dimension *dim* where the query box has been left or "infinity" if *nip* is inside the query box in dimension *dim*; initially the value is set to "infinity" for each dimension
  - `save_min`[*dim*] holds the step in dimension *dim* where the minimum of the query box has been exceeded; initially the value is set to -1 for each dimension
  - `save_max`[*dim*] holds the step in dimension *dim* where *nip* has fallen below maximum of the query box; initially the value is set to -1 for each dimension

  A simplified pseudo algorithm of such computation follows:

```
bp = maximal_bp;
while (bp > 0) {
      dim = get_dimension(bp);
      step = get_step(bp);
      if (z_value{bp} > lower_bound{bp}) {
            if (save_min[dim] == -1)
                  save_min[dim] = step;
      }
      else if (z_value{bp} < lower_bound{bp}) {
            if (flag[dim] == 0 && save_min[dim] == -1) {
                  out_step[dim] = step;
                  flag[dim] = -1;
            }
      }
      if (z_value{bp} < upper_bound{bp}) {
            if (save_max[dim] == -1)
                  save_max[dim] = step;
      }
      else if (z_value{bp} > upper_bound{bp}) {
            if (flag[dim] == 0 && save_max[dim] == -1) {
                  out_step[dim] = step;
                  flag[dim] = 1;
            }
      }
      bp--;
}
```

If `flag`[*dim*]=0 in all dimensions, then *nip* is actually the required intersection point. Otherwise the value of *nip* has to be corrected so that it lies inside the query box.

Because the nearest higher value than current *nip* is searched, at first the maximal bit position *max_bp* which has to be changed from 0 to 1 has to be determined. Let be *min_out_step*=min(`out_step`[*dim*]) and *min_dim* the corresponding dimension. Then two cases have to be distinguished:

- If `flag`[*min_dim*]=-1, then we have already found the bit position *max_bp*=*bp*(*min_dim*, *min_out_step*).
- If `flag`[*min_dim*]=1, then a higher *max_bp* has to be found because the bit position specified by *min_out_step* and *min_dim* has to be set to 0 - particularly a bit position that is lower than the bit position of `save_max`[*dim*] in corresponding dimension and with value equal to 0 is searched.

In both cases all bits following the *max_bp* has to be adapted according to rules mentioned in following pseudo code:

```
max_bp = bp(min_dim, min_out_step);
if (flag[min_dim] == 1) {
      max_bp = min({new_bp | new_bp > max_bp
            && get_step(new_bp) > save_max[get_dim(new_bp)]
            && z_value{new_bp} == 0 });
      // some attributes have to be updated for further processing
      save_min[get_dim(max_bp)] = get_step(max_bp);
      flag[get_dim(max_bp)] = 0;
}

// now the z-value can be changed accordingly in each dimension
foreach dimension dim {
      if (flag[dim] >= 0) {
            // nip has not fallen below the minimum in dim
            if (max_bp <= bit_position(dim, save_min[dim]))
                  "set all bits in dimension dim with
bit position < max_bp to 0 because nip would not surely get below
the lower_bound"
            else
                  "set all bits in dimension dim with
bit position < max_bp to the value of corresponding bits of the
lower_bound"
      }
      else
            // nip has fallen below the minimum in dim
            "set all bits in dimension dim to the value of
corresponding bits of the lower_bound because the minimum would not
be exceeded otherwise"
}
```

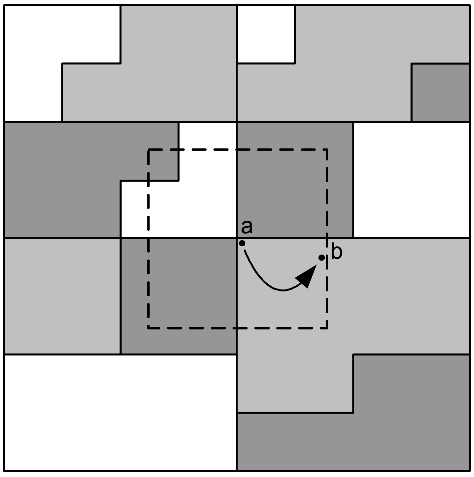
The overall time complexity of above algorithm can be easily estimated and equals to O(*n*) where *n* is the bit length of the maximal Z-value. The first part (comparison of *nip* with *lower_bound* and *upper_bound*) is of this complexity because of the while loop executed exactly *n* times. Then the possible searching for alternative *max_bp* has at most *n* steps. And the last foreach loop is executed *dim* times (*dim* < *n*) where each assignment of bits of *nip* within this loop is of constant complexity.

4.5.2 Get Next Z-value Out

Function *get_next_zvalue_out*(*z_value*, *lower_bound*, *upper_bound*) generates the nearest higher or equal Z-value (to the specified *z_value*) such that the Z-value that directly follows the result lies outside a query box determined by its *lower_bound* and *upper_bound* boundaries. In other words, for a point lying inside the query box it calculates Z-value *nlp* of the next leaving point with relation to the query box (see Figure 13).

Even though the tests has proven that dividing a query box into ideal Z-regions using this function is time consuming because of the nature of the Z-curve (see chapter 4.3.4 "Querying Tuples"), and therefore the only usage of this function is in procedural way of traversing the navigational type of the UB-Tree index which is not recommended because of high number of context switches (see chapter 4.4.7 "Querying Tuples in Procedural Way"), this algorithm is interesting and therefore it is described thoroughly in this work.

Figure 13: Example of starting point (a) and searched point (b) in get_next_zvalue_out function



The main idea is that for each dimension two numbers are counted - one is the minimal number that has to be added to the original *z_value* so that the result gets above the *upper_bound* in given dimension; on the contrary the other is the minimal number that has to be added to the original *z_value* so that the result gets below the *lower_bound* in given dimension. Finally the minimal one from all such numbers is chosen, it is decreased by one and is added to the *z_value*.

The algorithm exploits function *get_dimension*(*bp*) and operator {} which have been defined when describing the function *get_next_zvalue* (see chapter 4.5.1 "Get Next Z-value"). Similarly to *get_next_zvalue* several properties are held for each dimension which are being updated whilst original *z_value* is being processed bit by bit from the most relevant bit to the least relevant one according to the comparison of *z_value* on actual bit position with the value of *lower_bound* (or *upper_bound* respectively) on the same bit position.

The attributes that are being distinguished for each dimension *dim* are following:

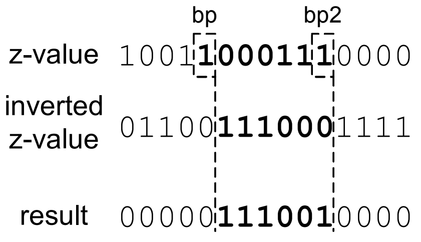
- `max_add`[*dim*] contains the value that has to be added to the *z_value* to get above the maximum of the query box in dimension *dim*; initially the value is set to 0 for each dimension.
- `min_add`[*dim*] contains the value that has to be added to the *z_value* to get below the minimum of the query box in dimension *dim*; initially the value is set to 0 for each dimension.
- `is_above`[*dim*] indicates whether the sum of current value of `max_add`[*dim*] and *z_value* is already above the maximum of the query box in dimension *dim*; initially the value is set to `false` for each dimension.
- `is_below`[*dim*] indicates whether the sum of current value of `min_add`[*dim*] and *z_value* is already below the minimum of the query box in dimension *dim*; initially the value is set to `false` for each dimension.
- `min_add_tmp`[*dim*] and `max_add_tmp`[*dim*] contain temporary values used for computation of `max_add`[*dim*] and `min_add`[*dim*]; initially the value is set to 0 for each dimension in both arrays.

  The usage of these temporary values is following: if a bit position *bp* is being processed and both *z_value*{*bp*} and *lower_bound*{*bp*} equal to 1 or both *z_value*{*bp*} and *upper_bound*{*bp*} equal to 0, then we know that 2^{bp} can be added to the *z_value* to get outside the query box, however there can be a lower bit position *bp2* where this rule is valid as well and therefore just 2^{bp2} can be added to *z_value*. These arrays thus contain the lowest currently known value.
- `bit_min_add`[*dim*], `bit_min_add_tmp`[*dim*], `bit_max_add`[*dim*] and `bit_max_add_tmp`[*dim*] contain a bit position that has to be changed in related values `min_add`[*dim*], `min_add_tmp`[*dim*], `max_add`[*dim*] and `min_add_tmp`[*dim*]; initially the value is set to -1 for each dimension in all arrays (i.e. no bit has to be changed).

  The point is that if we realize that we need to change a bit position *bp* in the original *z_value* from 0 to 1 to get above maximum or from 1 to 0 to get below minimum, then it is not wise to simply add 2^{bp} to the *z_value* because we may find a lower number than 2^{bp} that can be added to the *z_value* and still the bit on *bp* position will be changed provided that there is a lower bit position *bp2* equal to 1 in the *z_value*.

  Also the number that has to be added to *z_value* instead of 2^{bp} can be easily counted - it is an inverted value to the *z_value* between *bp-1* and *bp2* positions (i.e. all bits equal to 1 in *z_value* are set to 0 in the inverted z_value and vice versa) plus the inverted z_value on *bp2* position is set to 1. An example is shown in Figure 14.

Figure 14: Computation of inverted z-value


bp
bp2
z-value 1001**1000111**0000
inverted z-value 01100**111000**1111
result 00000**111001**0000

The function that sets all bits of a z_value *val* between a *bp* bit position and the last known bit position of *z_value* equal to 1 to the inverted *z_value* is called *set_inverse*(*val*,*bp*). For the proper computing we need to hold a temporary variable *last_one_bit* related to the last bit position of the *z_value* which equals to 1.

Some problems may come up during the computation:

- When trying to get below the minimum of the query box in a dimension and all following conditions evaluate to true:

  a) there are bit positions *bp* and *bp2* that both belong to the same dimension *dim*, i.e. *get_dimension*(*bp*) equals to *get_dimension*(*bp2*);

  b) both *z_value*{*bp*} and *lower_bound*{*bp*} equal to 0;

  c) all bits of *z_value* between *bp* and *bp2* equal to 1;

  d) we realize that 2^{bp2} should be added to either `min_add`[*dim*] or `min_add_tmp`[*dim*],

  then the bit on *bp* position in the sum of *z_value* and `min_add`[*dim*] would change from 0 to 1 and the result would still not be below the minimum of the query box in dimension *dim*.

  Therefore `min_add`[*dim*]{*bp*} has to be set to 1 so that it is again set back to 0 in the sum of *z_value* and `min_add`[*dim*].

  Thus we keep a temporary variable *last_zero_bit* that contains the bit position *bp* where the last 0 bit of the *z_value* has been processed, and in case it is necessary then all relevant bits of dimension *dim*=*get_dimension*(*bp*) in `min_add`[*dim*] or `min_add_tmp`[*dim*] are set to 1 by simple signifying that the `bit_min_add`[*dim*] or `bit_min_add_tmp`[*dim*] equal to *bp* instead of *bp2*.

  Please note that this problem does not arise when `bit_min_add`[*dim*] has already been set to a bit position *bp*. In such case the inverted *z_value* (between *bp* and a lower bit position *bpX* which is either equal or lower than currently processed bit position *bp2*) is about to be added to the original *z_value* and their sum will have all bits behind the *bp* bit position set to 0.

- When trying to get above the maximum of the query box in a dimension *dim* and all following conditions evaluate to true:

  a) *z_value*{*bp*} for a bit position *bp* equals to 1;

  b) *upper_bound*{*bp*} equals to 0;

  c) `min_add`[*dim*] is not equal to -1,

  then we may not simply signify that we are above the query box in the dimension, because the inverted *z_value* added to temporary result would change the bit on *bp* position in the sum from 1 to 0 and thus the result would not be above the maximum of the query box. In other words we have to do the same things as if both *z_value*{*bp*} and *upper_bound*{*bp*} were equal to 0.

  Similarly when both *z_value*{*bp*} and *upper_bound*{*bp*} equal to 1 and `min_add`[*dim*] is not equal to -1, then we have to do the same things as if *z_value*{*bp*} was equal to 0 and *upper_bound*{*bp*} was equal to 1.

Following pseudo code stands for the main loop of the algorithm:

```
last_zero_bit = maximal_bp;
last_one_bit = maximal_bp;
bp = maximal_bp;
while (bp > 0) {
      dim = get_dimension(bp);
      // compare z_value with lower_bound
      if (!is_below[dim]) {
            if (z_value{bp} > lower_bound{bp}) {
                  if (bit_min_add[dim] == -1) {
                        if (get_dimension(last_zero_bit) = dim)
                              bit_min_add[dim] = last_zero_bit;
                        else
                              bit_min_add[dim] = bp;
                  }
            }
            else if (z_value{bp} < lower_bound{bp})
                  is_below[dim] = true;
            else if (z_value{bp} == 1) {
                  if (get_dimension(last_zero_bit) = dim)
                        bit_min_add_tmp[dim] = last_zero_bit;
                  else
                        bit_min_add_tmp[dim] = bp;
            }
      }

      // remember the last_zero/one_bit
      if (z_value{bp} == 0 && get_dimension(last_zero_bit) != dim)
            last_zero_bit = bp;
      else if (z_value{bp} == 1)
            last_one_bit = bp;

      // compare z_value with upper_bound
      if (!is_above[dim]) {
            if (z_value{bp} < upper_bound{bp}) {
                  if (bit_max_add[dim] > -1)
                        set_inverse(max_add[dim], bit_max_add[dim]);
                  bit_max_add[dim] = bp;
            }
            else if (z_value{bp} > upper_bound{bp}) {
                  if (bit_max_add[dim] > -1)
                        bit_max_add_tmp[dim] = bp;
                  else
                        is_above[dim] = true;
            }
            else {
                  if (z_value{bp} == 0)
                        bit_max_add_tmp[dim] = bp;
                  else if (bit_max_add[dim] > -1) {
                        set_inverse(max_add[dim], bit_max_add[dim]);
                        bit_max_add[dim] = bp;
                  }
            }
      }

      bp--;
}
```

Some actions take place after the main loop:

- At first the function *set_inverse*() is called on all `min_*`[*dim*] and `max_*`[*dim*] values that have `bit_*`[*dim*]>-1.
- If `is_below`[*dim*] or `is_above`[*dim*] is still not true for a dimension *dim*, then the temporary result `min_add_tmp`[*dim*] has to be merged with `min_add`[*dim*], respectively `max_add_tmp`[*dim*] with `max_add`[*dim*].
    - Following logic applies for getting above maximum: let *bp* be the most relevant bit equal to 1 in `max_add_tmp`[*dim*], then all bits lower than *bp* are set to 0 in `max_add`[*dim*] and then `max_add_tmp`[*dim*] is added to `max_add`[*dim*].

      The reason for this process is that `max_add`[*dim*] currently contains a number that has to be added to *z_value* so that the sum of these numbers is not below the maximum of the query box in dimension *dim*, whilst `max_add_tmp`[*dim*] contains the minimal number that has to be added to get one of the bits of *z_value* in dimension *dim* above the *upper_bound*.

      Marginal condition applies when there is no bit equal to 1 in `max_add_tmp`[*dim*] - in such case the result for dimension *dim* is excluded from the final selection of minimum.
    - When trying to get below minimum and `bit_min_add_tmp`[*dim*] is not equal to -1, then:
        - ¤ `min_add`[*dim*] is directly considered to be the searched number if also `bit_min_add`[*dim*] is not equal to -1, because all bits behind `bit_min_add`[*dim*] bit position are set to 0 in the sum and therefore the sum is already below minimum;
        - ¤ `min_add`[*dim*] is replaced with `min_add_tmp`[*dim*] otherwise.

      The result for dimension *dim* is excluded from the final selection when `bit_min_add_tmp`[*dim*] equals to -1.
- Finally all the `min_*`[*dim*] and `max_*`[*dim*] numbers are compared and the lowest one is chosen.

  If all the numbers are excluded from the final selection because of the conditions mentioned in the previous step, then the maximal possible Z-value ("infinity") is returned as the result (so the algorithms employing this function have to handle this situation).

The overall time complexity of above algorithm can be easily estimated and equals to O(*n*) where *n* is the bit length of the maximal Z-value. The first part (comparison of *z_value* with *lower_bound* and *upper_bound*) is of this complexity because the while loop is executed exactly *n* times and each inner step within this loop (including the calling of *set_inverse*() function) is of constant complexity. All the actions at the end of the algorithm are executed at most 2**dim* times where *dim* is the count of dimensions, however *dim* < *n* and therefore this computation does not influence the overall time complexity.

4.5.3 Inside Box

Function *inside_box*(*z_value*, *lower_bound*, *upper_bound*) checks whether a multidimensional tuple represented by its *z_value* lies inside a query box determined by its *lower_bound* and *upper_bound* boundaries.

The algorithm simply utilizes the first part of the function *get_next_zvalue* (see chapter 4.5.1 "Get Next Z-value") where a *z_value* is compared bit by bit with both boundaries. Then the result can be generated according to the value of `flag`[*dim*] attribute mentioned in the description of *get_next_zvalue*. Time complexity is also O(*n*) where *n* is the bit length of the maximal Z-value.

CHAPTER 5

Experiments

5.1 Testing Environment

All experiments in general have to be well defined in advance before the actual testing: it is necessary to know what should be tested, where it should be tested, how the tests should look like and what the measured values are. The experiments within this work are focused on several properties related to multidimensional index methods. This section describes whole background of testing scenarios.

5.1.1 Database Systems & Examined Indexes

The main purpose of this work was the comparison of a relational UB-Tree index implementation with its native kernel integration. Since the relational index has been developed on Oracle DBMS [8], it is the main platform to be studied. The only well-known commercial database system providing UB-Tree natively is the Transbase [9], thus it is comprehended into the experiments as well. In addition, the standard compound B-Tree of Oracle is tested to see Oracle's default handling of multiple-attribute indexes.

All tests have been performed on one physical computer with AMD Athlon64 3200+ CPU, 1GB RAM and 120GB 7400rpm hard disk.

Table 1: Overview of Examined Index Types

DBMS	index type	identifier
Oracle 10g XE	native compound B-Tree	btree
	relational UB-Tree, direct scheme	direct
	relational UB-Tree, navigational scheme, recursive SQL traversing	navig_sql
	relational UB-Tree, navigational scheme, procedural traversing	navig_proc
	relational UB-Tree, navigational scheme, traversing via a cursor	navig_cursor
Transbase 6.4.2	native UB-Tree	transbase

Particular database systems and index methods are referred to during experiments according to identifiers defined in Table 1. In some cases all the methods of navigational scheme are referred all together with usage of "navig" identifier.

5.1.2 Data

All the experiments were evaluated on synthetic data generated just for the purpose of testing. Similarly to [7], several sets of data with size 10^4, 10^5, 5×10^5, 10^6 and 5×10^6 were prepared in advance. For each set, multidimensional points in dimensions 2, 3, 4, 5, 10, 15 and 20 were obtained from a random number generator so that influence of both the data size and dimension could have been studied. Range of a dimension was set to 2^{32} in all cases.

Concerning hypothetical real data, their entries usually do not fill whole space regularly, but they are usually formed into several clusters. Thus also the synthetic data were generated to create clusters and to correspond at least partially to real data. The count of clusters was dependent on the size of data according to definitions in Table 2.

Table 2: Count of Clusters in Dependence on the Database Size

#entries	#clusters
10'000	10
100'000	100
500'000	500
1'000'000	700
5'000'000	1000

Let *dc* be the count of dimensions, *rng* the range of dimensions, and *cc* the count of clusters. Then the width of a cluster *x* was determined according to following equation:

$$x = \frac{rng}{\sqrt[dc]{cc \times dc^2}}$$

5.1.3 Values to be Determined

Following values were recognized for each type of index:

- **Count of accessed pages**: the count of pages that were accessed during evaluation of a single DML statement is probably the most relevant information regarding the performance of an access method.

  Even though the actual performance is mostly influenced by the count of physically accessed disk pages in comparison with the count of pages accessed in memory (time to fetch a page from a disk is significantly higher than memory access time), the actual measured value stands for the sum of both disk and memory pages accessed during evaluation. The reason for this approach is that such value is independent both on the actual hardware configuration of a machine used for experiments (size of RAM) and on settings of particular DBMS (size of cache etc.).

- **Time**: in real applications, the factual time to access searched entries is also very relevant information. It is heavily platform and hardware dependent and thus it has only an informative value in the experiments. However, as all the tests were performed on one physical machine, we may consider the time related results to be adequate for comparison of examined methods.
- **Size of the index**: this value stands for the actual size of an index structure on a disk.

  In case of Oracle it can be determined from system catalogues; it is computed from the count of blocks that index occupies. E.g. in case of navigational scheme, many blocks may not be fully filled because of clustering and thus the real size of an index may be smaller. However, this information is not of high relevancy and although it can be also found out from system tables, it is not considered in the experiments.

  In case of Transbase the size of an index can be identified by running a statistical tool.

5.1.4 Methods

In all cases, the page size was set to 8KB so that the comparison of count of accessed pages would be relevant. In order to keep the conditions equivalent for all index types, both the data sets and query windows were prepared before testing. It means that queries related to all indexes were identical and performed upon the same data.

Moreover, all queries were modified to be in form "SELECT COUNT(*) FROM …" to avoid fetching time to be included in the overall performance time. It is likely that the COUNT aggregation is not a time-consuming operation and even though it was, it would influence the results in a similar way for all index types.

The query windows always covered some well defined part of whole space; particularly windows covering 0.1%, 0.5%, 1%, 5%, 10% and 30% of the space were generated for each dimension. For each such "selectivity" 20 random query windows were prepared and the results were then averaged.

For testing purposes a special application was prepared. It connected to a DBMS via ODBC interface, evaluated the queries and collected statistics either from system catalogue (Oracle) or from the output of a console application (Transbase). Then the results were transformed into the form which can be processed by the R-project application [12] which was used for creation of graphs.

5.2 Improvements Determined During Experiments

Implementation of a relational access method has to rely on underlying database engine. Therefore some approaches and techniques used during the programming part can be found unusable or just improper during experiments. This section discusses some problems which were identified during testing.

5.2.1 Optimal Constant for Extended Query Box in Direct Scheme

During the implementation of the relational UB-Tree via the direct scheme (see chapter 4.3 "UB-Tree via the Direct Scheme") a necessity to decompose the original query box into several continuous Z-regions arose. Because the count of such Z-regions is usually very high, this requirement was changed into decomposition of such called *extended query box* into set of Z-regions (see chapter 4.3.4 "Querying Tuples").

A constant defining the length of minimal Z-region which covers an extended query box must have been found. From the construction of the extended query box follows that such length equals to $(2^{\text{const}})^{\text{dim}}$, where *dim* is the count of dimensions of the space and *const* is a user-defined constant. Consequently, the extended query box exceeds (or drops below) the boundaries of the original query box by at most 2^{const}-1 in each dimension.

Therefore above problem is simplified into identification of "optimal" *const* number. The higher the *const* number is, the less Z-regions are used for decomposition of an extended query box but the more tuples are filtered out because they belong to the extended query box and not to the original query box. Thus a trade-off solution must be found.

Let *width* be the bit width of the longest range within a domain of the original query box (e.g. a query box in 3-dimensional space of size 500×500×1000 has its longest range 1000 and therefore its *width* equals to 10).

The experiments have shown that the *const* should be equal to *width*-4 in 2-dimensional space. The extended query box is then most often covered by 10 - 40 Z-regions; this number also stands for the count of range queries executed on the `ub_index` table. If we consider the fact that in real applications a query box is usually of similar width in each dimension, then we may estimate that its extended query box will get over the boundaries of the original box by approximately 6% of its width in a dimension, thus the count of tuples which are filtered out should be acceptable.

However, these findings cannot be so simply applied for higher dimensions. Following circumstances should be taken into account:

- Time complexity of the decomposition is linearly dependent on the count of such called *basic cubes* (see chapter 4.3.4 "Querying Tuples") which all together comprise the extended query box.

  Provided that the *const* is in general in form *width-x* where *x* is an integer, the extended query box can be covered by up to $(2^x+1)^{dim}$ Z-regions with the smallest allowed length (which equals to $(2^{const})^{dim}$). Even though the count of such Z-regions is for sure significantly higher than the real count of *basic cubes* (many of the cubes are of a higher length), it should not be disregarded. This is mostly true in high dimensions (15 or 20) where even in case of *x*=1, this number is too high.

- From previous point follows that it may be wise to keep *x*=0 for higher dimensions. However, high dimensionality causes that even query boxes with relatively small selectivity have their *width* very close or even equal to the bit width of the dimensions *dim_width* (please, keep in mind that all dimensions are considered to be of the same width). It is not wise to have the *const* equal to the *dim_width*, because the Z-region covering whole space would be always returned as the result of decomposition; note that its length equals to $(2^{dim\_width})^{dim}$.

  In such case we may consider a different point to estimate optimal *const*. The *const* does not even have to be an integer because the overall performance is sufficient if whole space is divided into reasonable number of Z-regions. The count of such Z-regions is $(2^{dim\_width\ -\ const})^{dim}$, which is quite high number even if the *dim_width-const* equals e.g. to 0.5 and the count of dimensions is 15 or 20.

In order to support above theory, Table 3 shows the optimal setting of *const* with respect to the count of dimensions. It also shows average count of Z-regions which arise from decomposition of a query box in given dimension, and approximated maximal exceeding over or below the boundaries of a query box in a dimension.

Table 3: Setting of Optimal Constant for Decompose Query Box Algorithm with Respect to the Count of Dimensions

#dimensions	const	average #z-regions	max exceed
2	*width* - 4	24	6%
3	*width* - 2	26	25%
4	*width* - 1	18	50%
5	*width* - 1	31	50%
10	*width* (≤ *dim_width* -1)	35	100%
15	*width* + 0.5 (< *dim_width*)	19	140%
20	*width* + 0.6 (< *dim_width*)	15	150%

5.2.2 Inserting Tuples in Navigational Scheme

Even though the proposed way of descending the UB-Tree via a recursive SQL statement seemed to be the most straightforward one in order to find a logical UB-Tree page where a tuple being inserted belongs during the implementation of the relational UB-Tree according to the navigational scheme (see chapter 4.4.3 "Inserting a Tuple"), the actual evaluation of recursive statements in Oracle is probably not the optimal one.

An alternative way to obtain the searched page is exploitation of the secondary database index defined on the *z_lower* attribute of the `ub_index` table which is used in case of procedural traversing through the UB-Tree (see chapter 4.4.7 "Querying Tuples in Procedural Way"). The actual statement to identify the page based on *z_value* of inserted tuple is then following:

```
SELECT * FROM ub_index x
WHERE
      x.z_lower >=
            // find nearest smaller tuple in the table
            (SELECT max(u.z_lower) FROM ub_index u
                  WHERE u.z_lower <= :z_value)
      AND x.z_lower <
            // find nearest upper tuple in the table
            (SELECT min(u.z_lower) FROM ub_index u
                  WHERE u.z_lower >= :z_value)
      AND x.level <= 1
      AND rownum = 1
```

The first sub-query of the statement determines either the nearest lower tuple on the level 0 or the nearest boundary of a Z-region on the level 1 (there can be actually some super Z-regions on higher levels with the same lower bound, however these are filtered out because of the condition "`x.level <= 1`"). Similarly the nearest higher tuple (or a boundary of a Z-region) is determined.

Only the first row fulfilling the criteria is chosen according to condition "`rownum = 1`" (a tuple on the level 0 can also stand for a boundary on higher level and thus more rows can get into the result). Consequently, the page identifier is extracted form the result according to following criteria:

- if the *level* of the resulting row equals to 0, then the *page_id* attribute of the row is the searched identifier
- if the *level* of the resulting row equals to 1, then the *son_id* attribute of the row is the searched identifier

Although the above statement with two sub-queries looks rather time consuming in comparison with the original recursive SQL query, the actual time for inserting set of tuples into the `ub_index` table is approximately 2-3 times faster in this case.

5.2.3 Using Optimizer Hints

Oracle provides developers with set of advanced tools that can be used to influence its behavior. At first, the query analyzer enables database users to see the actual execution plan for an SQL DML statement. If the execution plan does not seem to be the right one, optimizer hints can be used to force the database engine to use an alternative execution plan. Even though the plans determined by the database are quite often acceptable, evaluation of some statements may become a resource killer.

Hints for the optimizer are mostly required for DML statements that are used during inserts, updates and deletes of tuples. To speed up its performance, Oracle generates the execution plan for an SQL statement only once and stores it into its cache of plans along with a hash of given statement. Then, if the same statement is executed again (i.e. its hash can be found in the cache), Oracle reuses the execution plan. However, it seems that Oracle keeps the plan during whole session even when the count of items in a table changes significantly and therefore it would be wise to use a different plan.

This is obviously a problem when inserting thousands or millions of entries during a session. At first the database is empty, Oracle chooses an execution plan which often comprises full table scan (because it is really the fastest way when there are only few items in a table) and later it just reuses the plan because the statement is still the same.

Therefore it is wise to examine all DML statements that are used during the implementation and to find out their execution plans when all tables are empty. Hints for access paths should then be defined to ensure that the database chooses the proper plan.

In order to fulfill these findings, e.g. the SQL statement to obtain logical UB-Tree page mentioned in previous section needs to be rewritten in following way with usage of the optimizer hints:

```
SELECT /*+ INDEX(x idx_ub_index_z_lower)
      INDEX_SS(@lower_bound u idx_ub_index_z_lower)
      INDEX_SS(@upper_bound u idx_ub_index_z_lower) */
      * FROM ub_index x
 ...
            SELECT /*+ QB_NAME(lower_bound) */ max(u.z_lower)
            ...
      AND
            SELECT /*+ QB_NAME(upper_bound) */ min(u.z_lower)
            ...
      ...
```

The first hint refers to the outer condition on the *z_lower* attribute and recommends the usage of index `idx_ub_index_z_lower` defined on it. The other two hints refer to the inner sub-queries and again recommend usage of the same index, however in this case it should be used for a "skip scan" access (it means that optimal access according to min/max condition is chosen).

5.2.4 Page Size of UB-Tree and Cluster Definition in Navigational Scheme

An answer to the question "how many tuples should be stored in one logical UB-Tree page" is quite difficult. As the physical storage of table entries on a disk can be influenced only slightly in the relational approach (unlike the case when an index is integrated directly into database kernel and the disk management is implemented "on demand"), the only available improvement in contrast to a naive approach is clustering of items belonging to one logical page together.

A trade-off has to be identified between computation related to processing of one page and seeking to another page. The more items are stored in a page, the more computation is involved in the processing and the less seeking should be done. Experiments have shown that the optimal number is somewhere between 64 and 128 items in a page (both less and more items bring worse results).

So if a page size is chosen, a cluster size can be defined. At first it is necessary to know at least approximately the size of one row belonging to the cluster. E.g. in case of 2-dimensions with ranges 2^{32}, the average size is about 43 bytes (this number has been determined from system tables of Oracle). If we choose page size to be 128, it seems that the cluster size should be 5.5kB. However, the cluster size must be either a divider or a multiplier of physical page size of the database engine (i.e. either 8kB or 4kB). We may think that the optimal cluster size is thus 8kB, however setting it to 4kB brings better results! The reason is that pages are seldom fully occupied (usually less than 70% of their capacity is used), thus 4kB are most often sufficient and those few pages with more items are just partially stored in an overflow area.

5.2.5 Optimizing Disk Access Cost in Direct Scheme

As a range query according to the *z_value* attribute to obtain a sequence of z-values is the most common task in case of the direct scheme (see chapter 4.3.4 "Querying Tuples"), it is wise to order the `ub_index` table according to this attribute. However, Oracle does not allow creating of index organized tables based on user defined data types. Even if it allowed this, the performance of inserts into such table would be negatively impacted by often reorganization of the table.

This task can be simply achieved by recreating the table once all entries are stored in the database. A temporary table `ub_index_temp` is created via following DDL statement:

```
CREATE TABLE ub_index_temp
AS SELECT * FROM ub_index ORDER BY z_value
```

Then the original `ub_index` table is dropped and the temporary table is renamed back to `ub_index`.

5.3 Results

The results of experiments were focused mainly in following:

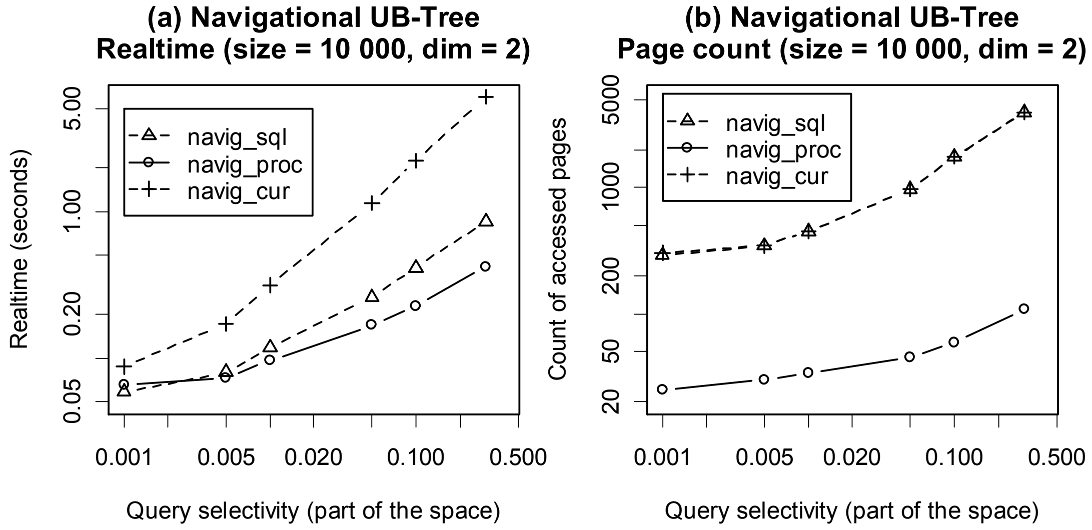
- identification of differences between particular methods used for traversing the UB-Tree structure (recursive SQL statement, procedural traversing and exploiting of a cursor) in case of the navigational scheme (see chapter 2.2.1 "Navigational Scheme of Index Tables");
- performance differences between the navigational scheme and the direct scheme (see chapter 2.2.2 "Direct Scheme of Index Tables");
- comparison of several properties of a relational UB-Tree index implementation in general with native implementation of both UB-Tree and compound B-Tree index.

5.3.1 Traversing the UB-Tree in Navigational Scheme

There are three ways of traversing the UB-Tree structure proposed for the evaluation of a range query in case of the navigational scheme (see chapter 4.4 "UB-Tree via the Navigational Scheme").

In Figures 15 - 18 their behavior is studied thoroughly; particularly the count of accessed pages and real time of query processing is determined in dependence on the selectivity of a query and the count of entries in the database. All mentioned results are related to 2-dimensional space. Behavior of the methods in higher dimensions is very similar, just all the measured values are higher because of the overall overhead caused by storing and processing more values.

Figure 15: Traversing UB-Tree in Navigational Scheme, 10'000 tuples



Each figure shows properties of the methods related to fix database size and comprises two graphs - the first one (*a*) shows the real time of query processing, whilst the second one (*b*) shows the count of accessed pages.

Common features concerning the count of accessed pages can be seen in all figures:

- The count of pages accessed in case of navig_sql and navig_cur is significantly higher than in case of navig_proc.

  The reason is probably an ineffective evaluation of a recursive SQL query by Oracle. If we investigate the execution plan for a recursive statement, we may realize that Oracle always involves full table scan in it. Actually, in the default execution plan of the statement used in navig_sql and navig_cur there are full table scans for each CONNECT BY clause, which causes even worse performance. With usage of optimizer hints we may achieve exploitation of an index in the CONNECT BY clause; however for some strange reason, Oracle accepts only standard index and not the cluster index, although the performance would surely be better. Moreover, one full table scan always persists (Oracle probably needs to collect some statistics about the table).

- The count of pages accessed in case of navig_sql is nearly the same as in case of navig_cur (it is a bit smaller for navig_cur but the difference is only slight and cannot be seen clearly in the figures).

  It means that there are usually only few Z-regions that lie completely inside a query box. Even for those Z-regions lying inside the box, the count of pages that are skipped because of interrupted traversing in the recursive SQL statement is similar to the count of pages that are physically accessed in secondary index on the *z_lower* attribute of the `ub_index` table when evaluating the range query in case of navig_cur (see chapter 4.4.8 "Querying Tuples via a Database Cursor").

Figure 16: Traversing UB-Tree in Navigational Scheme, 100'000 tuples

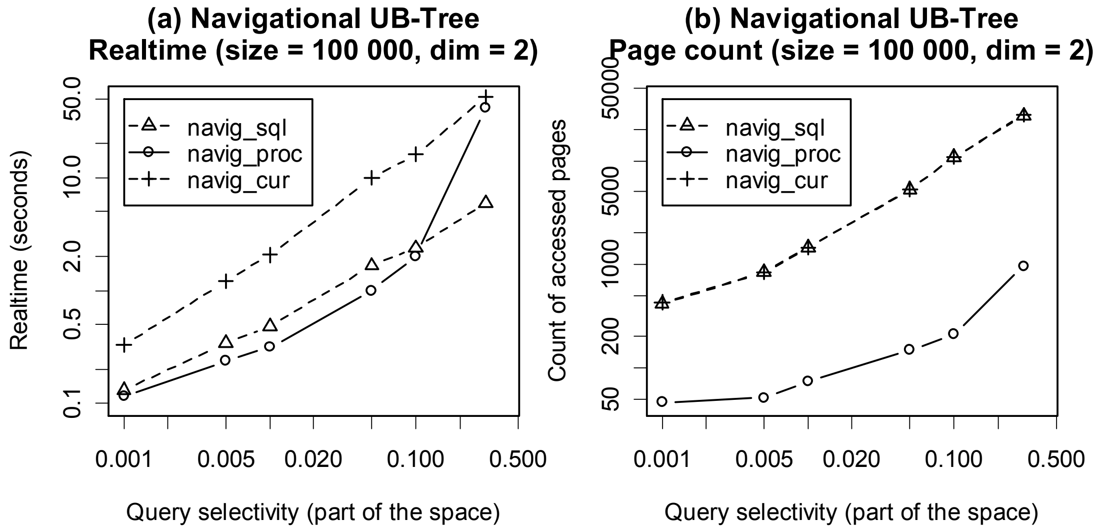


Figure 17: Traversing UB-Tree in Navigational Scheme, 500'000 tuples

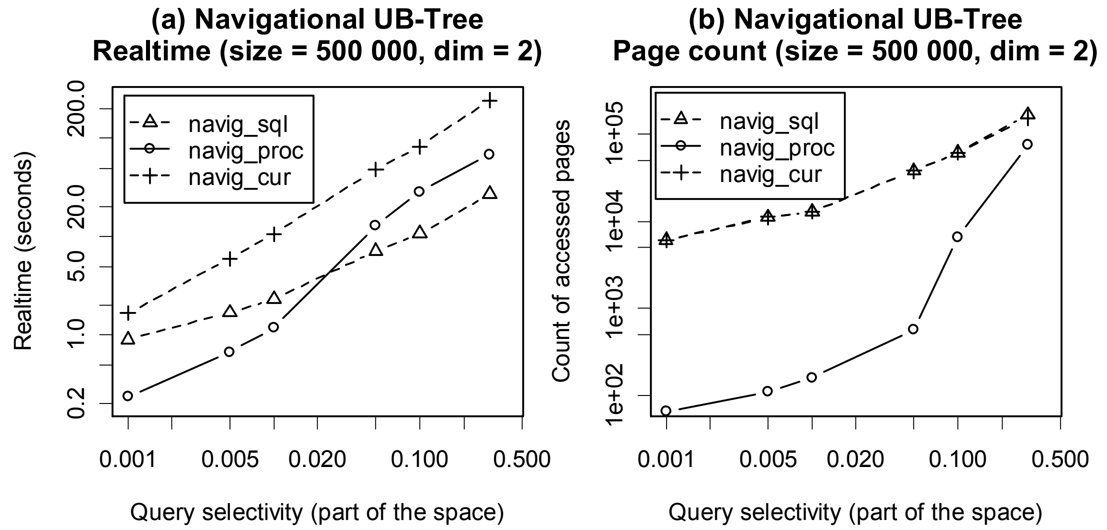
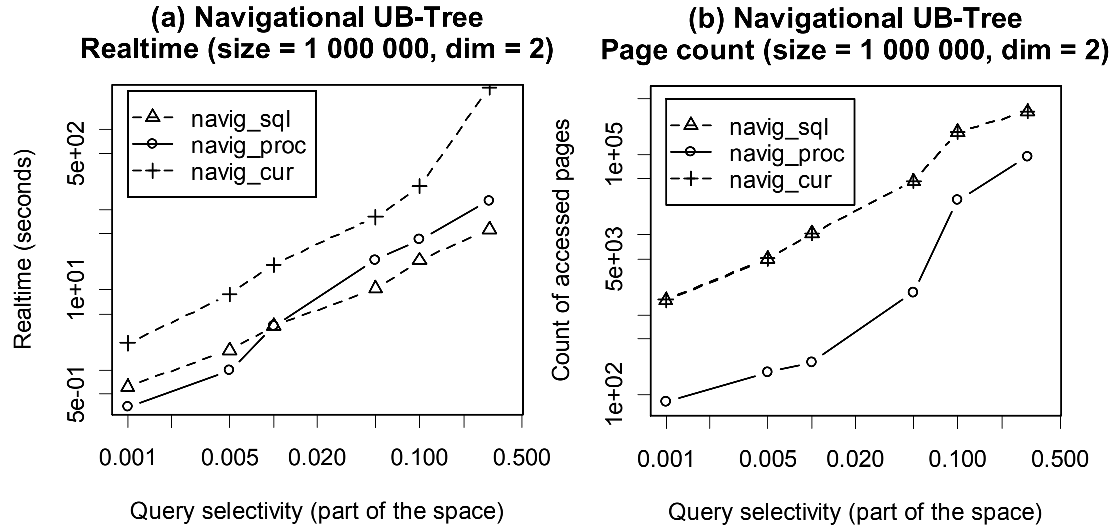


Figure 18: Traversing UB-Tree in Navigational Scheme, 1'000'000 tuples



Concerning the real time processing of queries, we may again realize some facts:

- Time for processing navig_cur is simply unbearable.

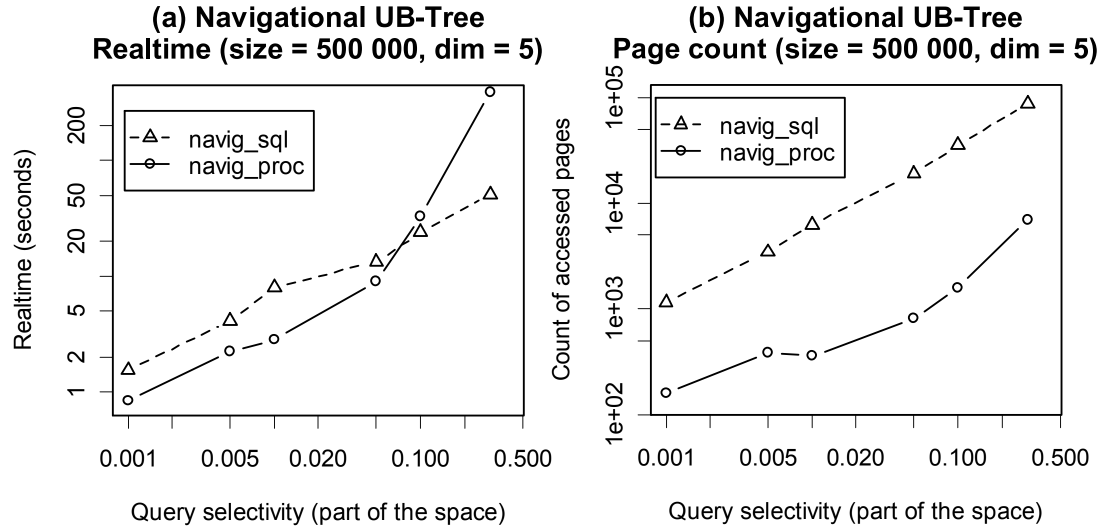
  Although the count of accessed pages is similar to navig_sql, the real time is much worse in case of navig_cur. The main reasons are probably both the overhead caused by holding opened cursor and high count of context switches for all Z-values and all Z-regions intersecting the query box.

- For a lower selectivity navig_proc takes less time than navig_sql, whilst for a higher selectivity navig_sql evaluates faster.

> The lower count of accessed pages (which is always much lower for navig_proc) may not result in faster processing. For higher selectivities the high count of context switches in navig_proc negatively influences overall behavior. On the other hand, for smaller selectivities the recursive procedural traversing is often stopped because many Z-regions do not intersect the query box and thus the number of context switches is kept relatively small. This approves the fact mentioned in both [1] and [2] that the more barrier crossings occur between database kernel and user defined code, the worse the performance is.

Similar dependences between navig_proc and navig_sql with respect to both the count of accessed pages and real time processing can be identified in higher dimensions as well. E.g. Figure 19 shows their behavior in case of 5 dimensions and 500'000 database entries:

Figure 19: Traversing UB-Tree in Navigational Scheme, 5 dimensions, 500'000 tuples



5.3.2 Direct Scheme vs. Navigational Scheme

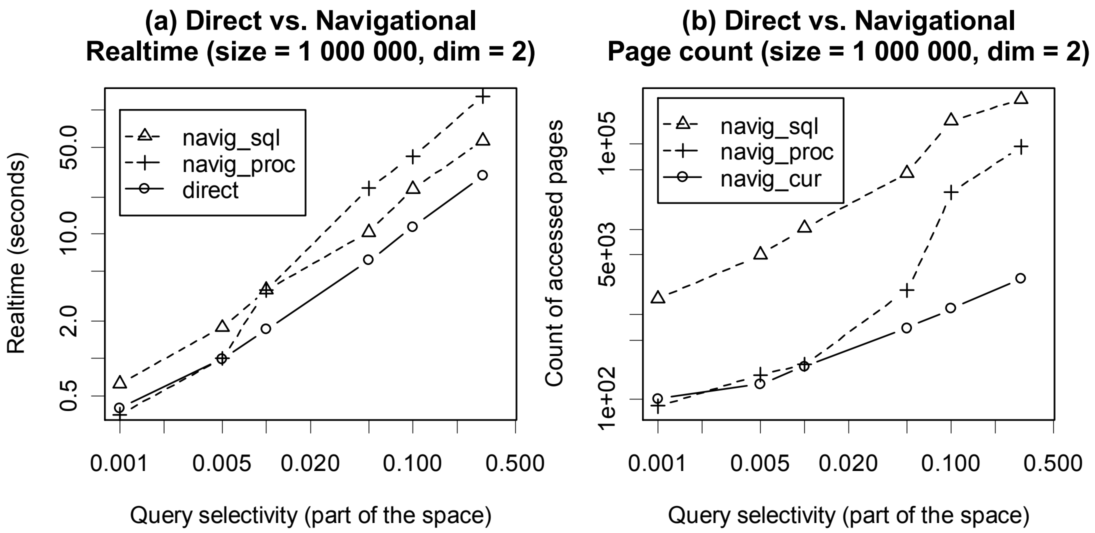
Considering the same criteria as in previous section, behavior of the navigational scheme (see chapter 4.4 "UB-Tree via the Navigational Scheme") and the direct scheme (see chapter 4.3 "UB-Tree via the Direct Scheme") is compared hereinafter. Concerning the navigational scheme, only the traversing via navig_sql and navig_proc are included into the results since the performance of navig_cur is very poor, as proven previously.

Exploring the results in 2-dimensional space (Figure 20), it seems that the direct scheme overpowers both the methods of the navigational scheme.

Similar results were identified also for different database sizes in 2-dimensional space, at least with respect to the count of accessed pages. With smaller database size and small selectivity, the real time for processing direct scheme queries is higher than in case of both navigational methods. The reason is that the decomposition of a query box in the direct scheme is computationally intensive task and negatively influences the total time in mentioned case.

It is worth to mention that the complexity of such decomposition is related just to the dimension of the space and the shape of particular query box. Thus the computation takes the same time, no matter how big the database is. This is different from other methods where the overall performance is influenced mainly by the database size and query selectivity.

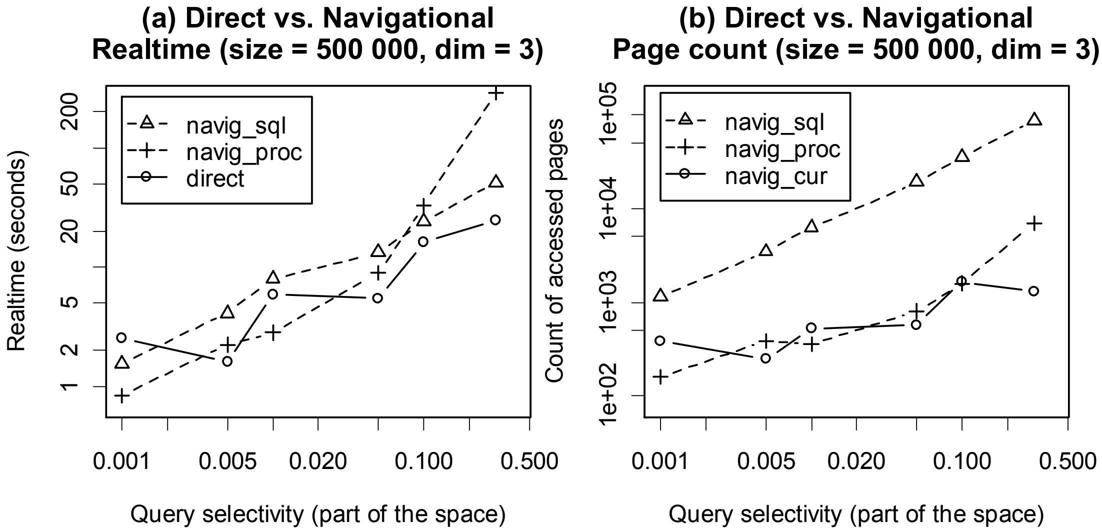
Figure 20: Comparison of Navigational Scheme and Direct Scheme, 2 dimensions, 1'000'000 tuples



Anyway, in Figure 21 we may realize that the behavior of the direct scheme could be really unpredictable. A remarkable finding is that in some cases a higher selectivity brings better results for both the real time processing and accessed pages count. Even though it cannot be simply proven, the reason is probably again the shape of a query box (or rather the actual count and size of Z-regions that arise from its decomposition).

The main drawback of chosen approach in the direct scheme is that the decomposition of a query box does not correspond to the physical storage of entries in the database at all. If a box is decomposed into several small Z-regions, interval queries on the `ub_index` table according to boundaries of such Z-regions may lead to traversing the existing secondary database index several times via the same physical path (i.e. with accessing the same disk pages several times). This is mainly obvious for Z-regions that are relatively close to each other and lie within an interval (super Z-region) which comprises small count of existing database items.

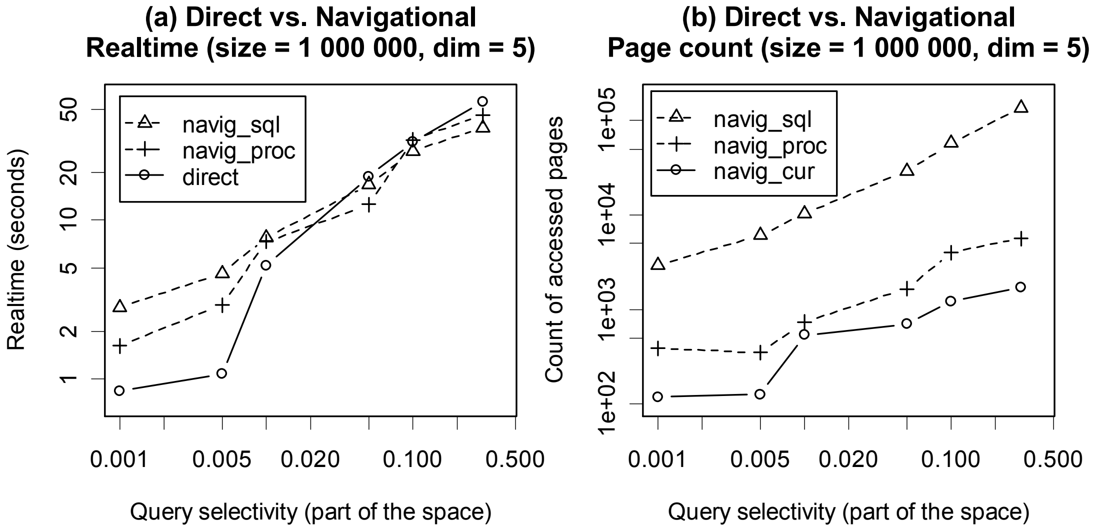
Figure 21: Comparison of Navigational Scheme and Direct Scheme, 3 dimensions, 500'000 tuples



(a) Direct vs. Navigational Realtime (size = 500 000, dim = 3)

(b) Direct vs. Navigational Page count (size = 500 000, dim = 3)

Figure 22: Comparison of Navigational Scheme and Direct Scheme, 5 dimensions, 1'000'000 tuples

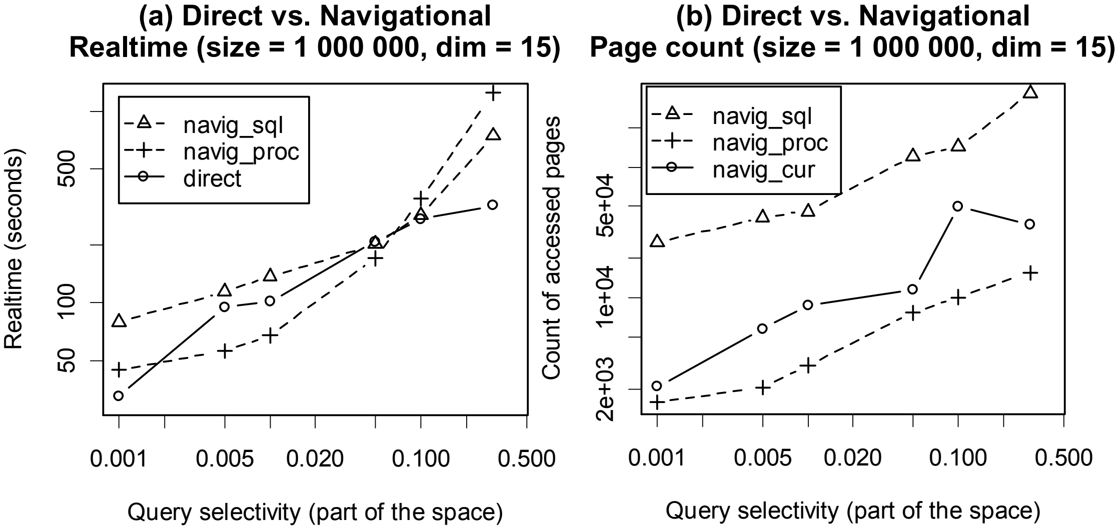


(a) Direct vs. Navigational Realtime (size = 1 000 000, dim = 5)

(b) Direct vs. Navigational Page count (size = 1 000 000, dim = 5)

Considering 5-dimensional space (Figure 22), it can be determined that a smaller count of accessed pages in the direct method may bring worse processing time with relation to the navig_proc, which contrasts to lower dimensions where the time is mostly in relation to the accessed pages count. This can be caused by higher cost of post filtering in the direct scheme. The more dimensions are involved, the bigger part of the space is covered by the extended query box in comparison with the original query box and thus more tuples are included into temporary result set and then filtered out.

Figure 23: Comparison of Navigational Scheme and Direct Scheme, 15 dimensions, 1'000'000 tuples



Negative impact of higher dimensions on the performance of the direct scheme can be also seen in Figure 23. In contrast to the 5-dimensional space, the count of accessed pages is always higher than in case of the navig_proc. If we take into account exceeding of an original query box by its extended query box by 140% in each dimension in 15-dimensional space both in upward and downward direction (see chapter 5.2.1 "Optimal Constant for Extended Query Box in Direct Scheme"), then the space covered by the extended box can be even many times bigger than the space covered by the original box.

An approach to overcome processing of such large spaces would be to generate an extended query box more precisely (i.e. an extended query box which would not exceed the original box so much). However, in such case the recursive computation takes quite a long time. Moreover, thousands of Z-regions are usually generated during decomposition, thus thousands of range queries are involved in such processing.

Unfortunately, neither of these approaches seems to be the appropriate one and the direct method remains far behind the optimal performance.

5.3.3 Index Size

Physical size of an index is one of the criteria that influence the performance of index evaluation, mostly with respect to the count of accessed physical pages. The bigger the index is the more pages are usually necessary to be processed.

Index size is more or less linearly dependent on both the size of the database and the count of dimensions (i.e. on the count of entries that are subject to index). However, this finding is only partial truth in case of used implementation of the relational index. The reason is the actual representation of Z-value, which is defined as an array of Oracle's NUMBER data type, which can hold up to 128-bit integer. As the range of a domain was always set to 2^{32} during the experiments, one item of such array can hold part of the Z-value for up to 4 dimensions. Thus the index size is quite similar for dimensions 1-4, 5-8, 9-12 etc. provided that the size of the database is the same.

Table 4 shows the dependency of index size on the size of database for all examined methods. Particularly, the results are valid for 4-dimensional space. All values are in megabytes.

Table 4: Physical Size (MB) of Index in Dependence on the Database Size

#entries	direct	navig	btree	transbase
10'000	1.1	2.7	0.5	0.5
100'000	11.2	26.7	5.1	4.9
500'000	51.2	12.0	23.5	24.0
1'000'000	100.3	242.0	47.1	47.9
5'000'000	484.3	1153.9	233.6	219.2

Following facts can be identified from the Table 4:

1. Index size is nearly the same for btree and transbase.
2. Index size is much bigger in case of relational methods.
3. Navig requires more than 2 times bigger space than direct.

All these facts can be simply explained if we look deeper on the amount of data stored for each index entry in particular method:

1. Both Oracle and Transbase store index data related to the same values (four 32-bit integers in case of 4-dimensional space) and build B+ Tree or UB Tree upon them. Thus the size of index differs only slightly based on actual representation of the tree and on more or less successful filling of physical disk pages with index entries.
2. Relational methods store nearly the same amount of data in relational tables as the standard methods store directly into disk pages. There can be some overhead to store user defined data types (Z-value) into database tables in case of relational methods, however this should not influence whole index size significantly.

The main reason for higher index size in relational methods is thus different. There are also standard secondary database index structures built upon the tables of a relational index to support performance of SELECT queries; and their size simply cannot be disregarded.

E.g. in case of direct method, the size of the index table takes approximately 45% of mentioned space, whilst the index built upon this table takes 55%. This finding also explains why the size of index in direct method is approximately twice higher then in case of btree or transbase methods.

3 A row of index table in navig method takes in average twice more space then a row in direct method in case of 4-dimensional space, because it simply holds more items. Thus the size of just the index table in navig method is similar to the size of whole relational index (table + related index) in direct method. Moreover, there are again some secondary database indexes defined on the navig index table which together take up to 60% of mentioned space.

5.3.4 Relational Index vs. Native Index Performance

Hereinafter, the performance of relational access methods is compared with access methods integrated natively into a DBMS kernel. The measured values are similar to previous cases (real time and count of accessed pages), however the procedure is different. Particularly, following relations are studied:

- Influence of dimension count on performance of all examined index methods (refer to Figure 24).
- Influence of database size on performance of all examined index methods (refer to Figure 25).

As we may see, for all methods both processing time and count of accessed pages grow more or less linearly in dependence on both the dimension count and the database size. Observing all the results more thoroughly, we may distinguish following:

1 Native kernel implementation of the UB-Tree in Transbase brings clearly the best performance with respect to both the count of accessed pages and real processing time.

2 Count of accessed pages measured in navig_proc and direct methods is usually smaller than in btree method. On the other hand, count of accessed pages in navig_sql method is always the worst one.

3 Despite of finding (2), the real processing time for all relational access methods is not only worse than native UB-Tree implementation in Transbase, but also several times worse than native implementation of "simple" compound B-Tree access method in Oracle.

Similar results can be identified also when different circumstances are taken into account than those ones presented in Figures 24 and 25. In other words, above findings are valid even for different selectivity, different database size (Fig. 24) or different dimension count (Fig. 25).

Figure 24: Impact of Dimension Count on Processing of Range Queries, 1'000'000 tuples, various selectivity

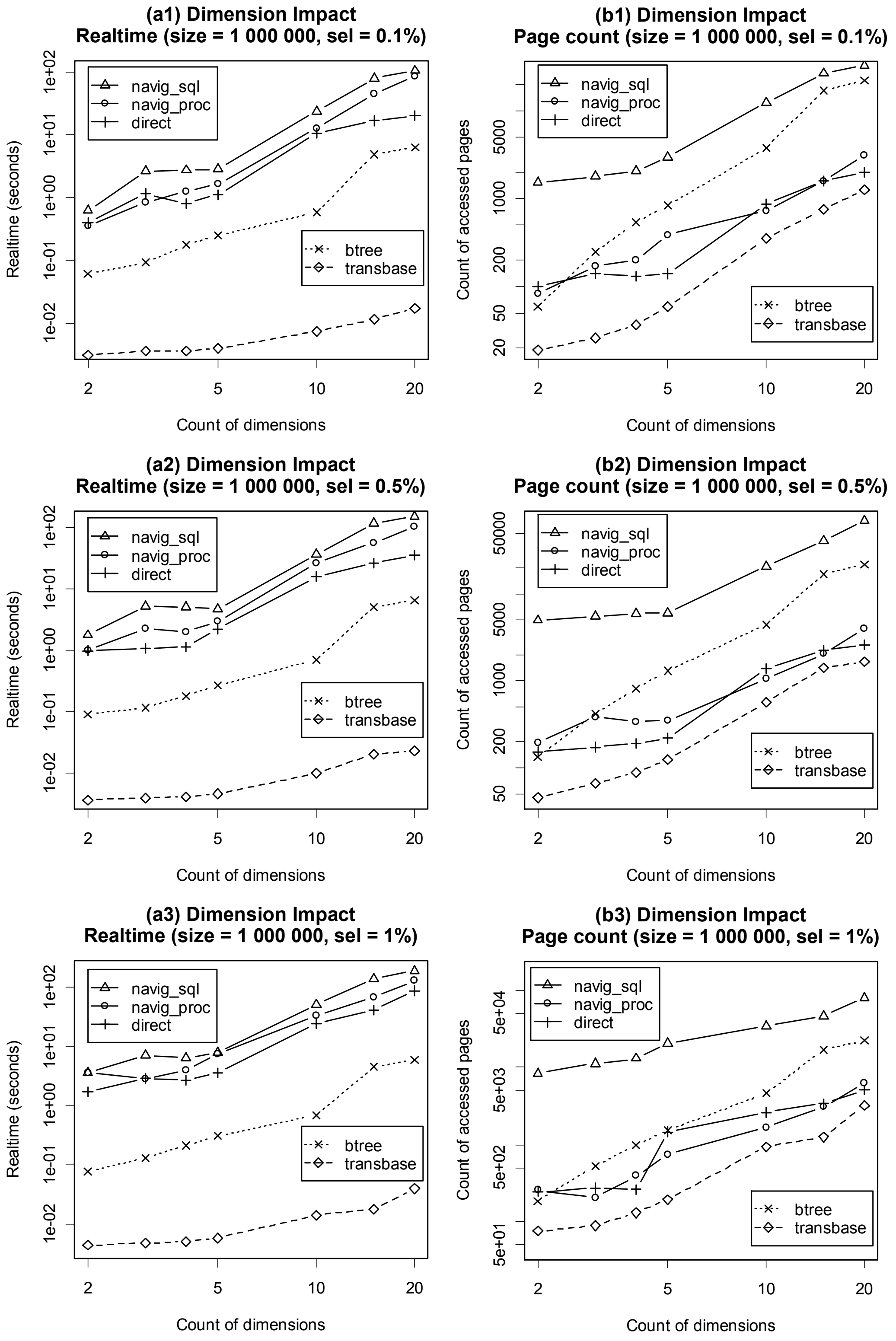
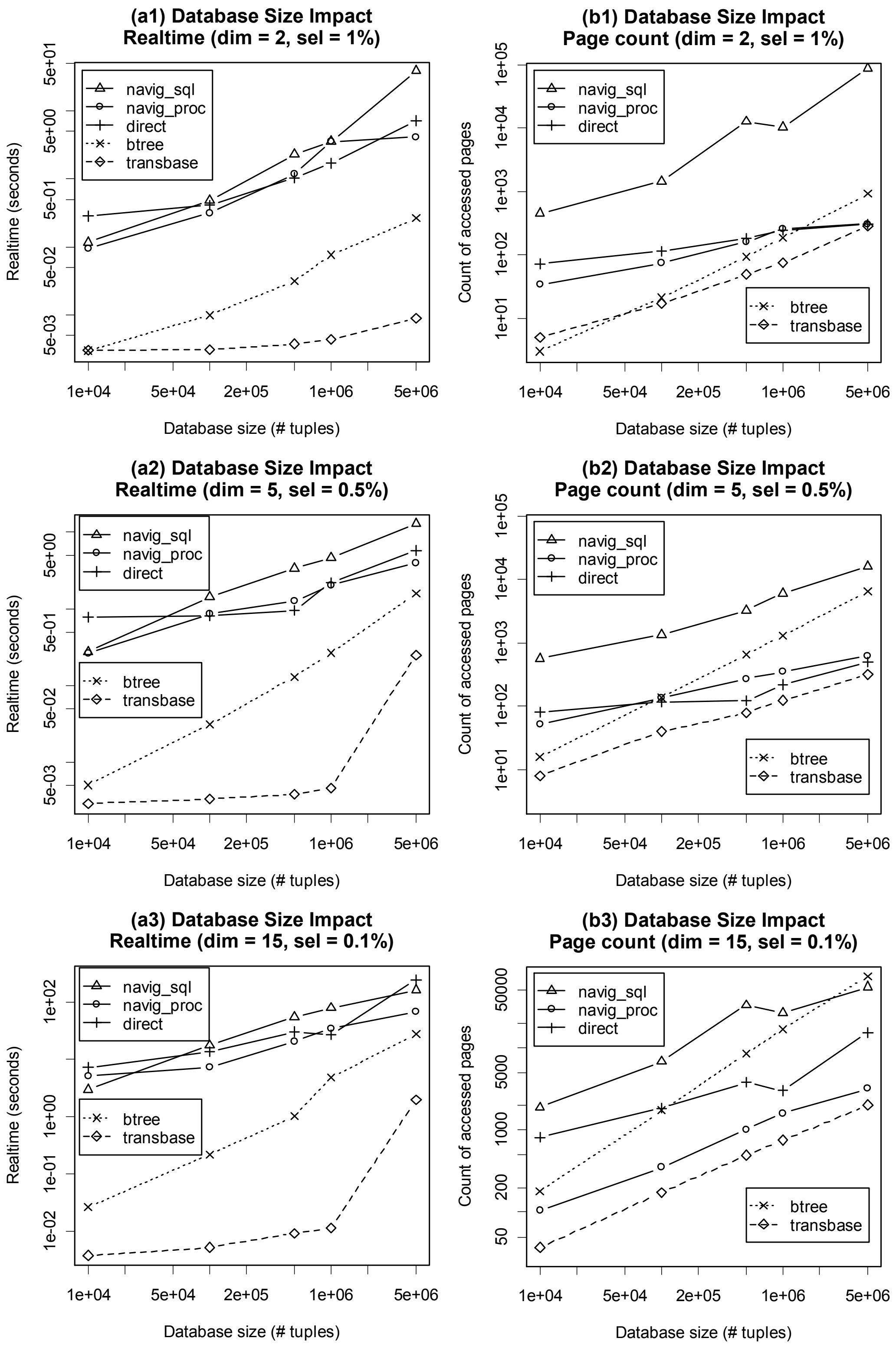


Figure 25: Impact of Database Size on Processing of Range Queries, various dimension count, various selectivity



Similarly to previous section, we will try to identify explanation or at least a deeper insight into each presented finding. Although many other causes can be considered, those ones listed below stand more or less for the main reason for determined behavior:

1 The excellent results of native UB-Tree implementation could have been expected without any doubts. In comparison with other methods it comprises set of advantages:

- UB-Tree access method is tailor-made for multidimensional data. Thus the performance of Oracle's native compound B-Tree is less efficient in all measured values.
- In contrast to any relational access method, native index implementation can handle any operation with most possible effectivity. This feature has already been mentioned in the Introduction and it just cannot be beaten simply.

2 The relational access methods generally bring some overhead into their evaluation. Secondary index structures have to be traversed and then the index table is searched on relevant items. However, in spite of this cost we may see that relational UB-Tree methods (particularly navig_proc and direct) usually require less disk pages to be accessed in contrast to the native compound B-Tree method. The difference is bigger with growing count of dimensions and also with growing database size. Thus we may see that the dedicated multidimensional access method again wins over the simpler one.

The bad performance of navig_sql has already been discussed previously. Full table scan that is always included into the execution plan of used recursive SQL query processing just causes that the count of accessed pages is too high.

3 The real time required for processing of a relational access method seems to be the dark side of its implementation.

Probably the most crucial condition that can negatively impact the overall behavior is the count of context switches involved in evaluation of user defined predicates and functions. All SQL statements in used implementation handle with custom data types `Type_tuple` and `Type_z_value` which are stored in index related tables.

Although Oracle provides the possibility to convert the PL/SQL code related to such data types and predicates into C, then compile the code into a dynamic library and then link the library, this procedure does not improve the performance much.

When thinking of the steps that the database engine has to take to evaluate access via each index type, we may approximate following simplified actions:

- In case of btree method, database kernel reads physical pages related to native compound B-Tree implementation and traverses them to the searched items.
- In case of direct method, quite heavy computation is involved in query box decomposition at the beginning of whole process; no disk pages are accessed yet, however the related time is considerable. Then the secondary index is used by the database kernel to access those pages of the index table that contain result candidates. Then all such candidates are filtered; the filtering comprises the mentioned overhead of evaluating user defined function from within a SQL statement.

- In case of navig_proc, high amount of context switches is comprehended directly into the way of traversing the index table, where several consequent SQL statements are issued from user environment to perform the actual evaluation of the access method.
- In case of navig_sql, the real processing time simply cannot be small when the count of accessed pages is significantly high.

All together, the native compound B-Tree overpowers the relational UB-Tree index implementation.

CHAPTER 6

Summary and Conclusion

In this work, implementation of a UB-Tree access method via the relational approach has been presented and compared with implementation of the UB-Tree directly into a database kernel.

As expected, the performance of a relational access method is far behind the performance of a native kernel integration of the same access method. The size of relational index is higher, queries with usage of native method require less disk pages to be accessed and the overall processing time is simply incomparable.

With respect to determined findings it seems that the native kernel integration of a new access method is the only suitable approach. However, this way is mostly available only to those developers who have access to source code of low level DBMS kernel functionality. Moreover, in terms of development time, the relational implementation is much cheaper since there is no impact of direct kernel changes on overall DBMS performance which may simply occur during the integrating approach. Thus it depends on several circumstances which way is the proper one to be chosen.

Usage of UB-Tree access method as a dedicated method for handling multidimensional data was proven to be more suitable than usage of less sophisticated B-Tree with compound keys. Considering just the count of I/O operations, this finding is valid even when comparing the relational UB-Tree index and a native implementation of compound B-Tree. Despite this fact, the time for processing queries with usage of the relational UB-Tree was found significantly worse than in case of the kernel evaluation of B-Tree access. The bottleneck of a relational approach is the necessity to use user defined functions in executive SQL queries.

Surely, usage of the relational approach is often the only available way when a requirement to build an index upon a custom data type arises. On the other hand, when native database data types are subject to index, it does not seem to be wise to incorporate a relational approach method which self uses custom data types. An interesting case would occur if a relational index could be implemented with usage of just the native database types, or at least when there were no custom predicates and functions in relevant SQL queries.

References

[1] Hans-Peter Kriegel, Martin Pfeifle, Marco Pötke, Thomas Seidl: *The Paradigm of Relational Indexing: A Survey*, Proc. 10. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW), Leipzig, 2003, in: Lecture Notes in Informatics (LNI), Springer, 2003

[2] Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhardt, Rudolf Bayer: *Integrating the UB-Tree into a Database System Kernel*, in: Proceeding of the 26th International Conference on Very Large Databases, Cairo, Egypt, 2000

[3] Robert Fenk, Volker Markl, Rudolf Bayer: *Interval Processing with the UB-Tree*, in: Proceedings of the 2002 International Symposium on Database Engineering & Applications, 2002

[4] Michal Krátký, Tomáš Skopal: *Benchmarking the UB-tree*, in: Proceedings of DATESO, Desná-Černá Říčka, 2003

[5] Rudolf Bayer. *The Universal B-Tree for multidimensional indexing: General Concepts*, in: Proceedings of World-Wide Computing and its Applications '97, 1997

[6] Tomáš Skopal, Michal Krátký, Václav Snášel, Jaroslav Pokorný: *On Range Queries in Universal B-trees*, ARG technical report, ARG-TR-01-2003, Department of Computer Science, VŠB-Technical University of Ostrava, 2003

[7] David Hoksza: *Multidimensional Indexing for Relational Databases*, diploma thesis (in Czech), Faculty of Mathematics and Physics, Charles University in Prague, 2006

[8] http://www.oracle.com/ … Oracle DBMS homepage

[9] http://www.transaction.de/ … Transbase DBMS homepage

[10] http://commons.wikimedia.org/wiki/Image:B-tree-aggregated-example.png

[11] http://en.wikipedia.org/wiki/PL_SQL … Details about Oracle's PL/SQL programming language

[12] http://www.r-project.org/ … Homepage of R-project, free software environment for statistical computing and graphics

[13] http://www.wampserver.com/en/ … Integrated package of Apache webserver with PHP5 engine and MySQL database for Windows platform

APPENDIX A

Relational UB-Tree Usage

This Appendix stands for a short guide of the relational UB-Tree index integration into Oracle DBMS. It also describes the procedure to launch benchmarking tests and reveals the content of the enclosed DVD disk.

Enclosed DVD Content

The content of the enclosed DVD is grouped into following folders:

- `installs`

  This folder contains installation files of these applications for Windows platform:

  - Oracle XE 10g
  - Oracle SQL Developer
  - Transbase 6.4.2

- `ub_tree`

  This folder contains executive SQL scripts for integration of relational UB-Tree into Oracle DBMS. Particularly, they are following:

  - `create_common.sql`: create SQL script containing definitions of functions, procedures, packages, data types and other database objects related to both the direct scheme and the navigational scheme implementations of relational UB-Tree.
  - `create_direct.sql`: create SQL script containing definitions of database objects related just to the direct scheme implementation of relational UB-Tree.
  - `create_navigational.sql`: create SQL script containing definitions of database objects related just to the navigational scheme implementation of relational UB-Tree.
  - `drop_common.sql`: drop SQL script which is used for destruction of database objects related to both the direct scheme and the navigational scheme implementations of relational UB-Tree.
  - `drop_direct.sql`: drop SQL script which is used for destruction of database objects related just to the direct scheme implementation of relational UB-Tree.
  - `drop_navigational.sql`: drop SQL script which is used for destruction of database objects related just to the navigational scheme implementation of relational UB-Tree.

- `benchmark`

  This folder contains .zip file with PHP scripts to connect to both Oracle and Transbase DBMS via ODBC interface, to run benchmarking tests and to collect results of the tests. There are also several subfolders containing pre-generated data of different size and dimension that are used during the tests.

Installation

Following steps have to be taken to integrate the relational UB-Tree into Oracle DBMS:

1 Install Oracle XE 10g (available on enclosed DVD disk). Connect as SYSTEM user to the Oracle and create a standard user that will be used as an account to log in to the database in order to integrate the relational index. This user should be granted all rights except from DBA to be able to integrate the relational UB-Tree.

If you plan to evaluate benchmarking tests under specified user as well, you need to grant also DBA right to this user, otherwise it is not possible to obtain specific information from system catalogues during the test.

2 For higher convenience, install Oracle SQL Developer (just extract relevant .zip file from enclosed DVD disk).

3 Run Oracle SQL Developer, connect to the database with user created in step (1) and load required scripts:

- To enable navigational scheme of relational UB-Tree, run scripts `create_common.sql` and `create_navigational.sql`.
- To enable direct scheme of relational UB-Tree, run scripts `create_common.sql` and `create_direct.sql`.
- To drop either of the schemes or whole relational index, run scripts `drop_navigational.sql`, `drop_direct.sql`, and eventually `drop_common.sql`. Please note that in case a navigational scheme is loaded into Oracle and you need to load the direct one, the navigational should be dropped at first (and vice versa).

4 Bind the index with a table as outlined in following section.

This step is not needed if you plan just to launch the benchmarking PHP application as described later, because it is done automatically during the benchmark tests.

Binding the Index with a Table

To bind the relational UB-Tree index with a table, please take following steps. Each step also comprises an example of its usage in 3-dimensional space:

1 Create a table that will contain primary data which are subject to the relational UB-Tree index.

```
CREATE TABLE primary_table (
      id NUMBER PRIMARY KEY,
      column_1 NUMBER,
      column_2 NUMBER,
      column_3 NUMBER
);
```

2 Define the constraints for each item (dimension) of the space.

```
INSERT INTO ub_constraints VALUES (1, -1000, 5000);
INSERT INTO ub_constraints VALUES (2, 0, 350000);
INSERT INTO ub_constraints VALUES (3, 1500, 3000);
```

3 Define appropriate AFTER TRIGGERs to keep the index data consistent with primary data.

```
CREATE TRIGGER ai_primary_table
      AFTER INSERT ON primary_table
      FOR EACH ROW
      BEGIN
            insert_tuple(Type_tuple(:new.column_1, :new.column_2,
                  :new.column_3), :new.id);
      END;
/
CREATE TRIGGER au_primary_table
      AFTER UPDATE ON primary_table
      FOR EACH ROW
      BEGIN
            update_tuple(Type_tuple(:new.column_1, :new.column_2,
                  :new.column_3), :new.id);
      END;
/
CREATE TRIGGER ad_primary_table
      AFTER DELETE ON primary_table
      FOR EACH ROW
      BEGIN
            delete_tuple(:old.id);
      END;
/
```

4 To exploit the relational UB-Tree in a range query, adjust a SELECT statement to employ the *inside_query_box* function.

```
SELECT primary.*
FROM TABLE(inside_query_box(Type_tuple(-500, 500, 2000),
      Type_tuple(4000, 4000, 2500))) index
LEFT JOIN PRIMARY_TABLE primary
      ON index.id = primary.id
```

Experiments Launching

In order to run auxiliary script to evaluate benchmarking tests, following steps are needed:

1 Install a webserver with PHP5 support. E.g. on Windows platform, free Apache 2.0 with pre-configured PHP5 can be downloaded from [13].

2 In `php.ini` configuration file of PHP, set the *max_execution_time* variable to 0 so that the tests are not interrupted untimely. Restart the webserver.

3 Define the ODBC data source connection for Oracle DBMS (eventually for Transbase DBMS) in your system settings.

4 Copy and extract the content of `benchmark` folder from the enclosed DVD to the executive folder of your webserver.

5 In the executive folder of your webserver, edit the copied file `settings.php` appropriately according to the tests you are about to launch and according to your DBMS and ODBC settings. It is necessary to set the benchmarked index type and log in information for your database connection by assigning proper values to *$database* variable within this file:

- `$database = new OracleDirect(`*odbc_name*`,` *username*`,` *password*`)` to benchmark the relational UB-Tree index with direct scheme of index tables;
- `$database = new OracleNavig(`*odbc_name*`,` *username*`,` *password*`)` to benchmark the relational UB-Tree index with navigational scheme of index tables;
- `$database = new OracleCompound(`*odbc_name*`,` *username*`,` *password*`)` to benchmark the native compound B-Tree index in Oracle;
- `$database = new Transbase(`*odbc_name*`,` *username*`,` *password*`)` to benchmark the native UB-Tree index in Transbase.

When benchmarking the Transbase DBMS, additional steps are needed:

- edit the file `tbstat.bat` so that it contains proper paths to `tbstat32` executable and `tbstatis.dat` file in your Transbase installation folder;
- run the executable `tbadm32` in your Transbase installation folder from system command line in the following way:

  `tbadm32 -i` *database_name* `monitor`

  The monitoring application has to be launched during whole benchmarking process to be able to collect statistical information from Transbase.

6 To perform the actual benchmarking tests simply type in your browser link to the `profile.php` script under the hostname assigned to your webserver (e.g. http://localhost/profile.php).

The script connects via ODBC to the specified database, evaluates the tests and writes the output to a file in your webserver executive folder.

To launch benchmarking tests with different settings just take steps (5) and (6) of above proceeding.

APPENDIX B

Relational UB-Tree Data Types

Short overview of user-defined data types that were used during the relational UB-Tree implementation is introduced in this Appendix. For each of them a simple description and particular SQL DDL statement used for its creation are presented.

Type_tuple

```
CREATE TYPE Type_tuple AS VARRAY(64) of NUMBER;
```

This data type represents a multidimensional tuple and serves as an interface between user tables and index tables. All index related functions and procedures that are used in both AFTER TRIGGERs and SELECT statements operate with this data type.

Type_numbers

```
CREATE TYPE Type_numbers AS TABLE of NUMBER;
```

This data type represents a temporary table of identifiers of primary data (which are required to be of NUMBER data type) and stands for the return value of the function *inside_query_box*() which is used in SELECT statements exploiting the relational UB-Tree index.

Type_z_value

```
CREATE TYPE Type_z_value_tuple AS VARRAY(32) of NUMBER;
CREATE TYPE Type_z_value AS OBJECT (
      z_value Type_z_value_tuple,
      ORDER MEMBER FUNCTION zval_order
            (other IN Type_z_value) RETURN NUMBER,
      MEMBER FUNCTION to_string RETURN STRING DETERMINISTIC
);
```

Data type `Type_z_value` is an object that represents the base element of the relational UB-Tree implementation - the Z-value. For its definition an auxiliary data type `Type_z_value_tuple` is needed; it is intended for holding the actual data of a Z-value.

The object comprises two member functions:

- Function *zval_order*() is used for comparison of two objects of given type, for example in ORDER BY SQL clause or in an algebraic comparison.
- Function *to_string*() generates a deterministic string from actual Z-value which can be used in a secondary functional index on a table column which is of `Type_z_value` type.

In fact, Type_z_value comprises also other auxiliary member functions and procedures than those listed. However, they are closely bound to the factual UB-Tree implementation and thus they are not of high relevancy for this overview. Should you be interested in the implementation details, please refer to the source code of the relational UB-Tree on enclosed DVD.

Type_boundary_table

```
CREATE TYPE Type_boundary AS OBJECT (
      lower Type_z_value,
      upper Type_z_value
);
CREATE TYPE Type_boundary_table AS TABLE OF Type_boundary;
```

Data type Type_boundary_table represents a temporary table of boundaries of Z-regions which arise from the decomposition of a query box in case of the direct scheme implementation of the relational UB-Tree. It is used as the return value of the function *decompose_query_box*() and for its definition an auxiliary data type Type_boundary is needed; it stands for an item (a row) of given table.