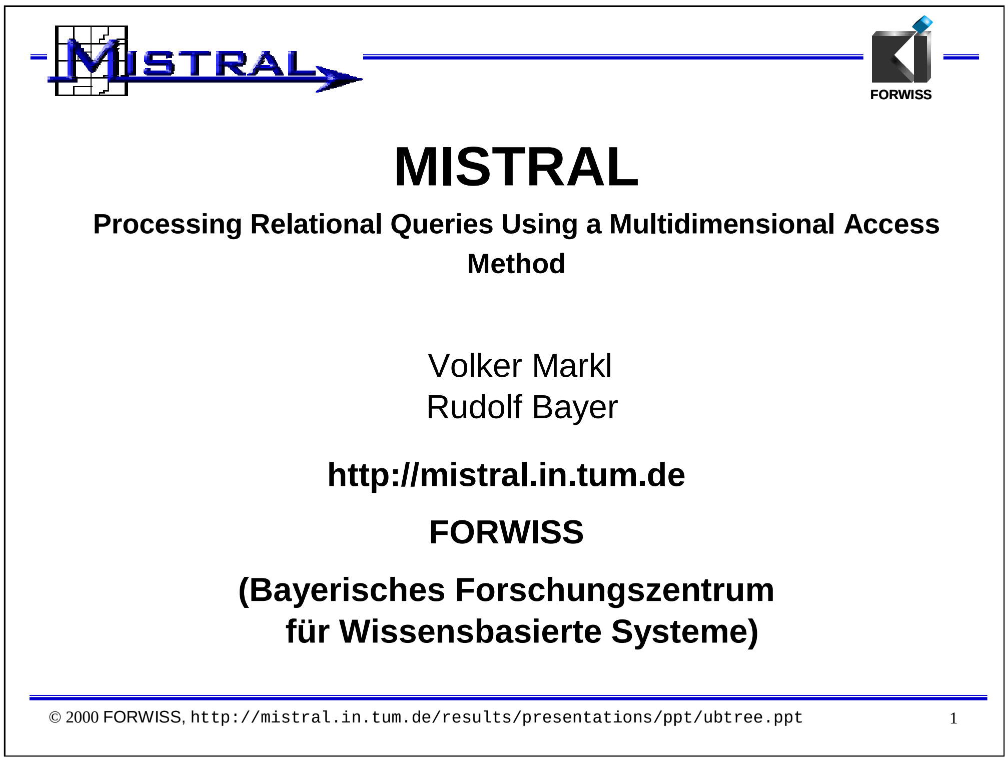


MISTRAL

Processing Relational Queries Using a Multidimensional Access Method

Volker Markl
Rudolf Bayer

http://mistral.in.tum.de

FORWISS

(Bayerisches Forschungszentrum für Wissensbasierte Systeme)

This Presentation consists of several animations and is mostly compiled out of the Ph.D. defense presentation of Volker Markl. Many parts and fragments have also been presented at several workshops and conferences (including VLDB 2000, ICDE 1999, IDEAS 1999) as well as at several DBMS vendors in U.S., Germany, and Japan.

The UB-Tree is a multidimensional access method that has been invented by Rudolf Bayer in 1996 and has been investigated by Volker Markl and Rudolf Bayer since then.

The presentation gives an overview of the MISTRAL project, an international research and development project with the goal to investigate the UB-Tree and its applications which has been undertaken at FORWISS from 1997 to 2000.

Further references, information, animations and interactive visualization tools can be found under URL http://mistral.in.tum.de

For detailed explanation of UB-Trees please refer to

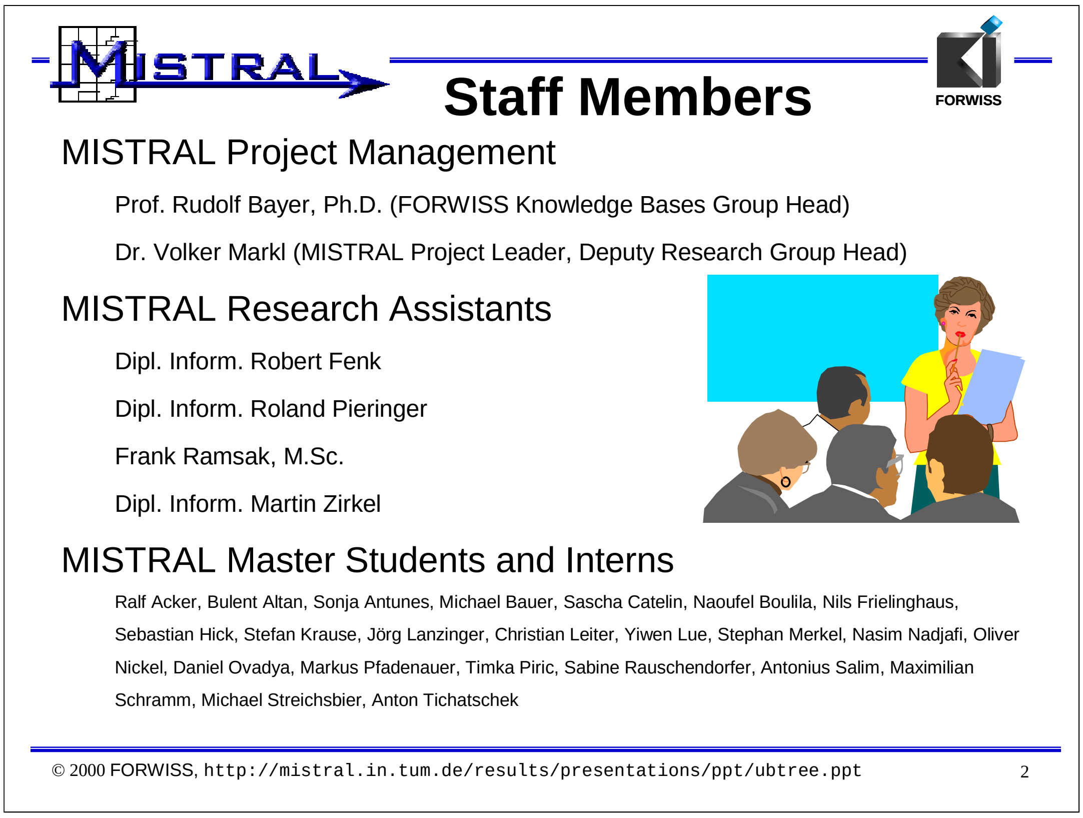
[Bay96] R. Bayer. The Universal B-Tree for multidimensional Indexing. Technical Report TUM-I9637,November 1996. http://mistral.in.tum.de/results/publications/TUM-I9637.pdf

[Mar99] V. Markl. MISTRAL: Processing Relational Queries using a Multidimensional Access Technique, Ph.D. Thesis, TU München, 1999, published by infix Verlag, St. Augustin, DISDBIS 59, ISBN 3-89601-459-5, 1999. http://mistral.in.tum.de/results/publications/Mar99.pdf

[MZB99] V. Markl, M. Zirkel, and R. Bayer. Processing Operations with Restrictions in Relational Database Management Systems without external Sorting. Proc. of ICDE Conf., Sydney, Australia, 1999. http://mistral.in.tum.de/results/publications/MZB99.pdf

[MRB99] V. Markl, F. Ramsak, and R. Bayer. Improving OLAP Performance by Multidimensional Hierarchical Clustering. Proc. of IDEAS Conf., Montreal, Canada, 1999. http://mistral.in.tum.de/results/publications/MRB99.pdf

[RMF+00] F. Ramsak, V. Markl, R. Fenk et al. Integrating the UB-Tree into a Database System Kernel Proc. of VLDB Conf. 2000, Cairo, Egypt, 2000. http://mistral.in.tum.de/results/publications/RMF+00.pdf



Staff Members

MISTRAL Project Management

Prof. Rudolf Bayer, Ph.D. (FORWISS Knowledge Bases Group Head)

Dr. Volker Markl (MISTRAL Project Leader, Deputy Research Group Head)

MISTRAL Research Assistants

Dipl. Inform. Robert Fenk

Dipl. Inform. Roland Pieringer

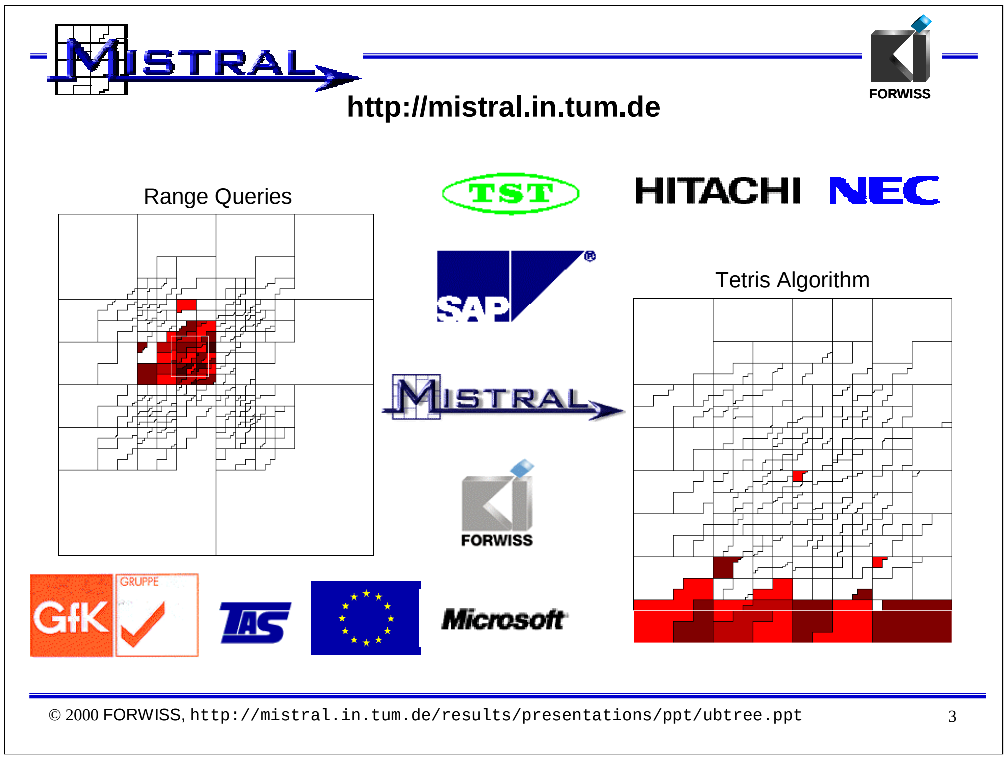
Frank Ramsak, M.Sc.

Dipl. Inform. Martin Zirkel

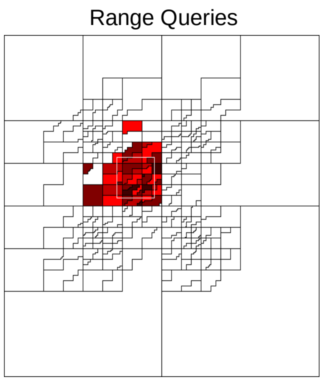
MISTRAL Master Students and Interns

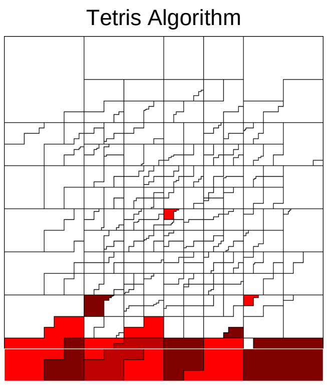
Ralf Acker, Bulent Altan, Sonja Antunes, Michael Bauer, Sascha Catelin, Naoufel Boulila, Nils Frielinghaus, Sebastian Hick, Stefan Krause, Jörg Lanzinger, Christian Leiter, Yiwen Lue, Stephan Merkel, Nasim Nadjafi, Oliver Nickel, Daniel Ovadya, Markus Pfadenauer, Timka Piric, Sabine Rauschendorfer, Antonius Salim, Maximilian Schramm, Michael Streichsbier, Anton Tichatschek

The MISTRAL project team consists of 5 full time researchers and a large set of master students, interns and guests of the participating companies.



http://mistral.in.tum.de

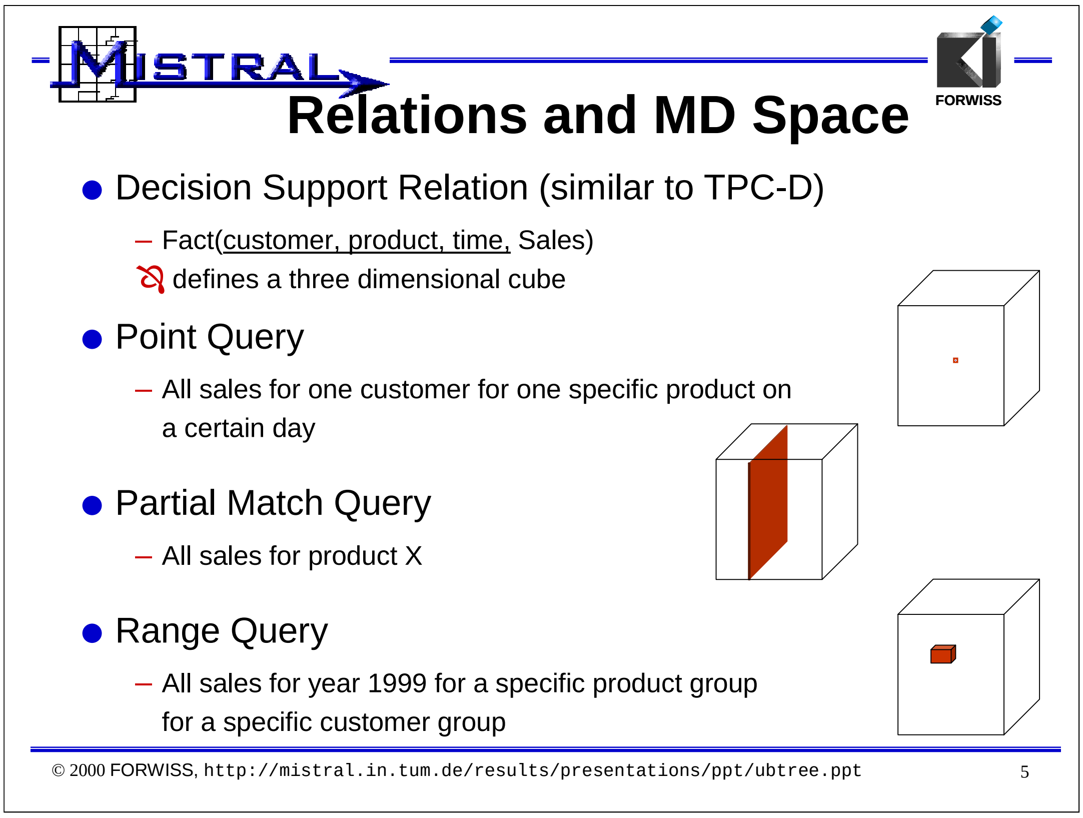




The partners on this slide have been sponsoring the MISTRAL work.

Overview

1. Concept of the UB-Tree: Z-Regions
2. Insertion
3. Range Query Algorithm
4. Tetris Algorithm
5. Kernel Integration
6. Performance Overview

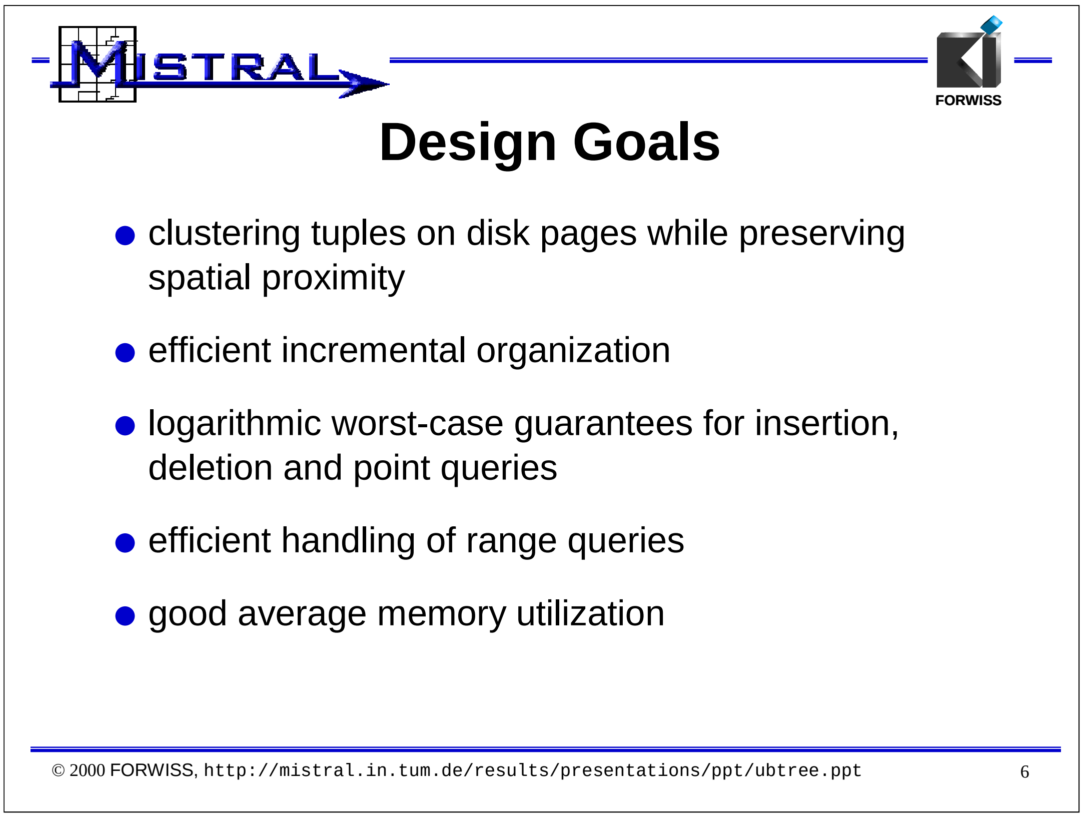


Relations and MD Space

- Decision Support Relation (similar to TPC-D)
  - Fact(customer, product, time, Sales)
  - defines a three dimensional cube
- Point Query
  - All sales for one customer for one specific product on a certain day
- Partial Match Query
  - All sales for product X
- Range Query
  - All sales for year 1999 for a specific product group for a specific customer group

© 2000 FORWISS, `http://mistral.in.tum.de/results/presentations/ppt/ubtree.ppt` 5

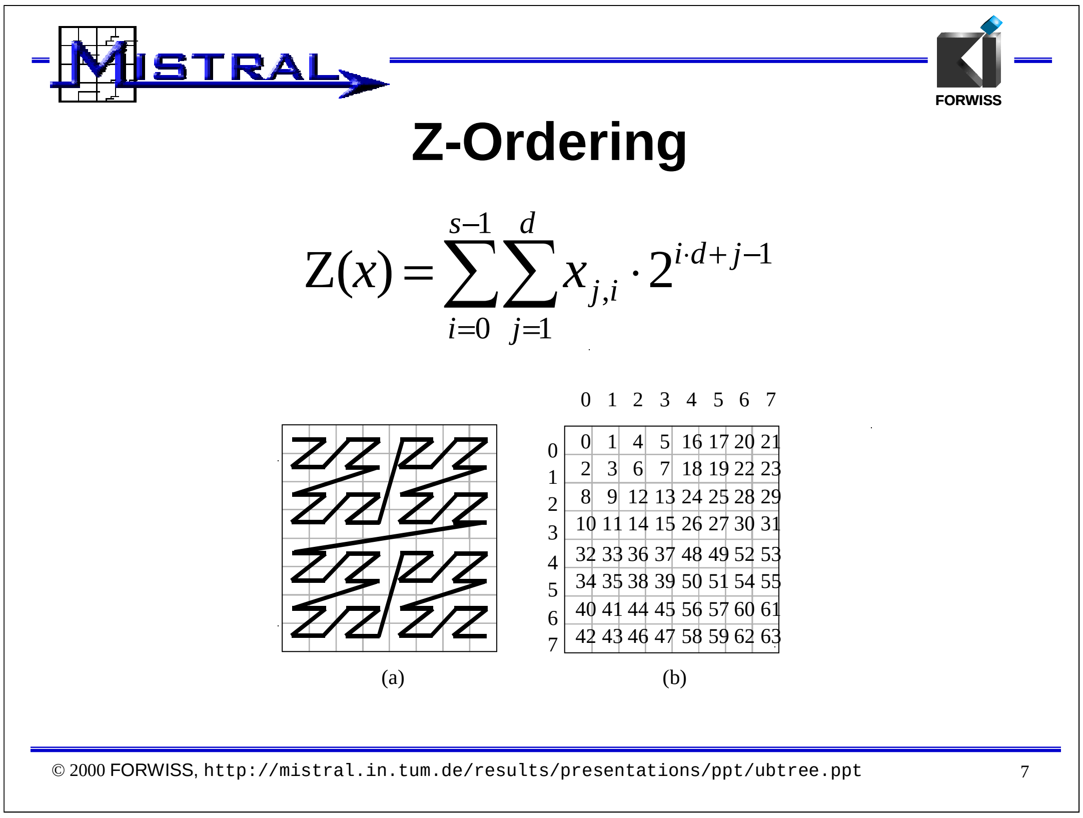
This slide gives an example of how to map queries with multi-attribute restrictions on a relational table to multidimensional range queries. Queries with multi-attribute restrictions are typical for decision support applications, but also frequent in archiving, geographical applications and relational database systems in general.



Design Goals

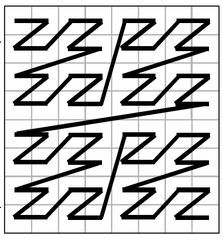
- clustering tuples on disk pages while preserving spatial proximity
- efficient incremental organization
- logarithmic worst-case guarantees for insertion, deletion and point queries
- efficient handling of range queries
- good average memory utilization

The design goals on this slide are generic for designing an access method, both for single-attribute or multi-attribute access methods. However, most multi-attribute methods do not fulfill all of the design goals. R-Trees, for instance, do not offer a good average memory utilization. Most multidimensional access methods like R*-Trees, Grid-Files or kd-B-Trees do not allow for efficient incremental organization, e.g., by requiring forced reinsertion, Grid-Splits or even complex reorganizations. Since UB-Trees (as we will see) rely on standard B-Trees for their storage organization of so-called Z-regions, they inherit all of the above properties from the underlying B-Tree structure.



Z-Ordering

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



(a)

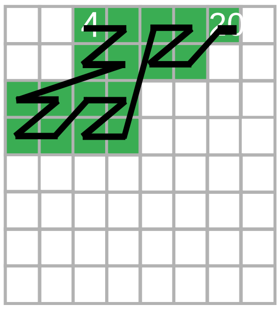
	0	1	2	3	4	5	6	7
0	0	1	4	5	16	17	20	21
1	2	3	6	7	18	19	22	23
2	8	9	12	13	24	25	28	29
3	10	11	14	15	26	27	30	31
4	32	33	36	37	48	49	52	53
5	34	35	38	39	50	51	54	55
6	40	41	44	45	56	57	60	61
7	42	43	46	47	58	59	62	63

(b)

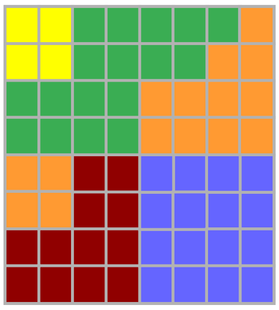
Z(x) is a bijective function that computes for every tuple x its Z-address, i.e., its position on the space filling Z-curve. The slide presents the Z-addresses (or Z-values) for an 8x8 universe. Z-values are efficiently computed by bit-interleaving as described e.g. by Orenstein and Merret in 1984. An additional animation `http://mistral.in.tum.de/results/presentations/ppt/zaddress.ppt` on the Mistral Web Site describes bit-interleaving.

Z-regions/UB-Trees

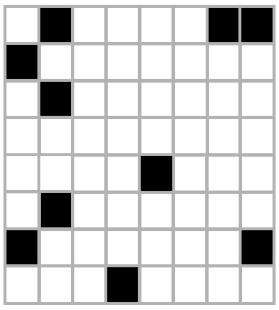
A *Z-region* [α : β] is the space covered by an interval on the Z-curve and is defined by two Z-addresses α and β.



Z-region [4 : 20]



UB-Tree partitioning: [0 : 3],[4 : 20], [21 : 35], [36 : 47], [48 : 63]



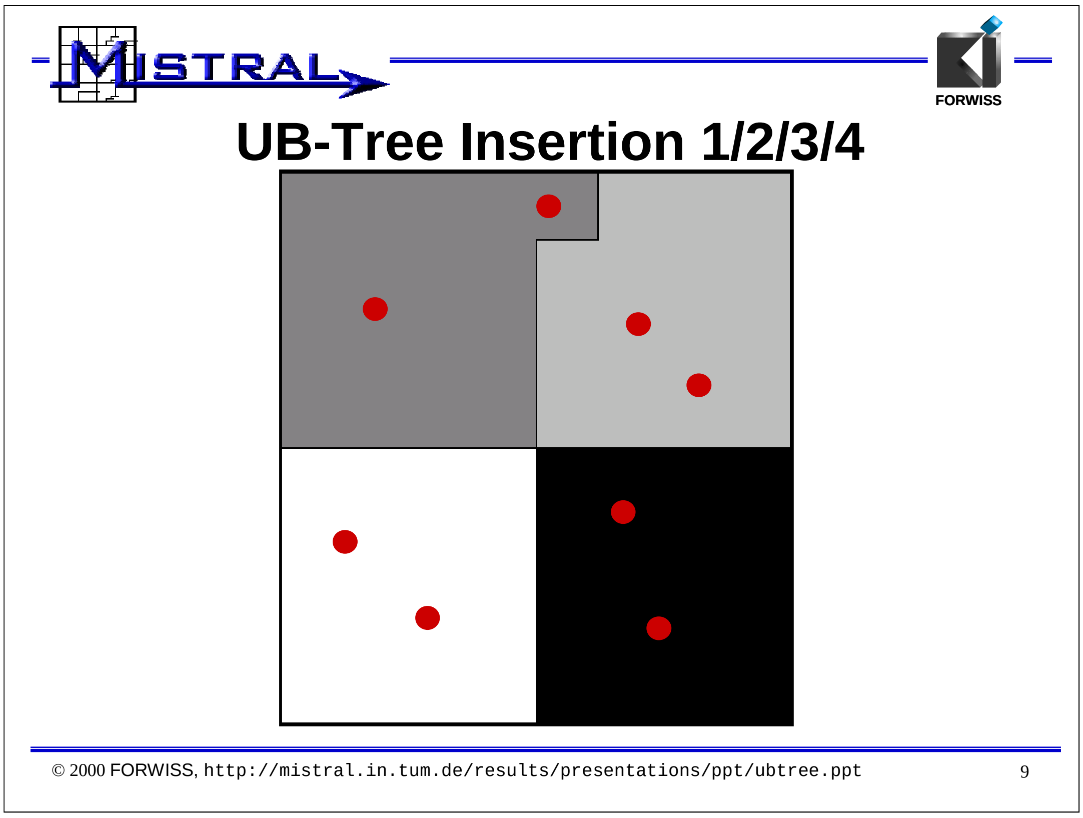
point data creating the UB-Tree on the left for a page capacity of 2 points

A Z-region is the space covered by an interval on the Z-curve. Thus a Z-region has two meaningful interpretations, a linear interpretation as an interval as well as a spatial interpretation. The left part of this animation shows the Z-region [4:20] and its spatial interpretation. The spatial extent of the Z-region becomes clearer if we draw the Z-region into the picture. Please note that bit-interleaving is an efficient means to calculate the Z-value for a tuple (or the inverse, i.e., the tuple values for a given Z-value). Thus we can arbitrarily switch between the linear Z-space and the geometric interpretation.

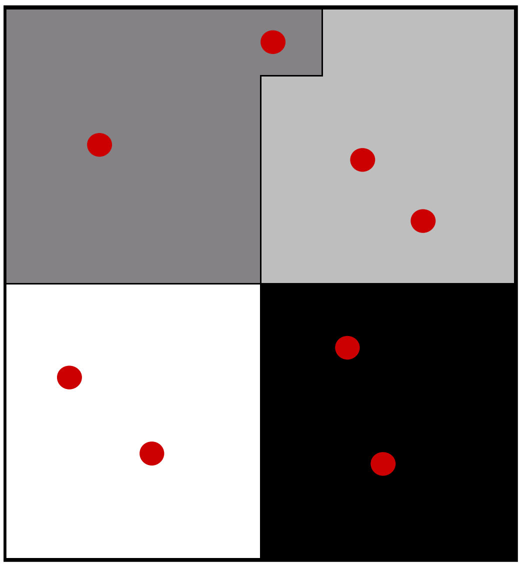
The middle part shows a Z-region partitioning (or also called UB-Tree partitioning) which is a disjoint set of Z-regions whose union covers the entire multidimensional space. In this picture the partitioning consists of 5 Z-regions. Most Z-regions preserve spatial proximity, i.e., neighboring points of a given point are in the same region with a high probability. The orange region [21 : 35] consists of two disconnected parts. If a Z-region could consist of many disconnected parts, this would prevent Z-regions from being suitable for clustering. However, [Mar99] gives a proof that regardless of the dimensionality of the Z-ordered space (i.e., not only for 2d) the number of not connected parts of a Z-region is at most two.

We can consider a Z-region corresponding to a disk page, i.e., being a container of a fixed or variable capacity storing tuples which fall into the spatial extent of the Z-region. The right part of the animation shows a point distribution of 10 points which with a page capacity of two points per page might be stored in the Z-region partitioning of the middle picture of this slide .

We use a B-Tree to store the upper limit of the Z-value of each Z-region and call the corresponding B-Tree storing the Z-region organization Universal B-Tree (UB-Tree) [Bay96, Mar99]



UB-Tree Insertion 1/2/3/4



© 2000 FORWISS, `http://mistral.in.tum.de/results/presentations/ppt/ubtree.ppt` 9

UB-Tree disk pages correspond to Z-regions. Each tuple is stored on a disk page corresponding to the Z-region that this tuple spatially belongs to. Each Z-region can store a certain capacity of tuples and must be split into two Z-regions during insertion if the page capacity of a Z-region is exceeded. Details can be found in [Bay96] and [Mar99]

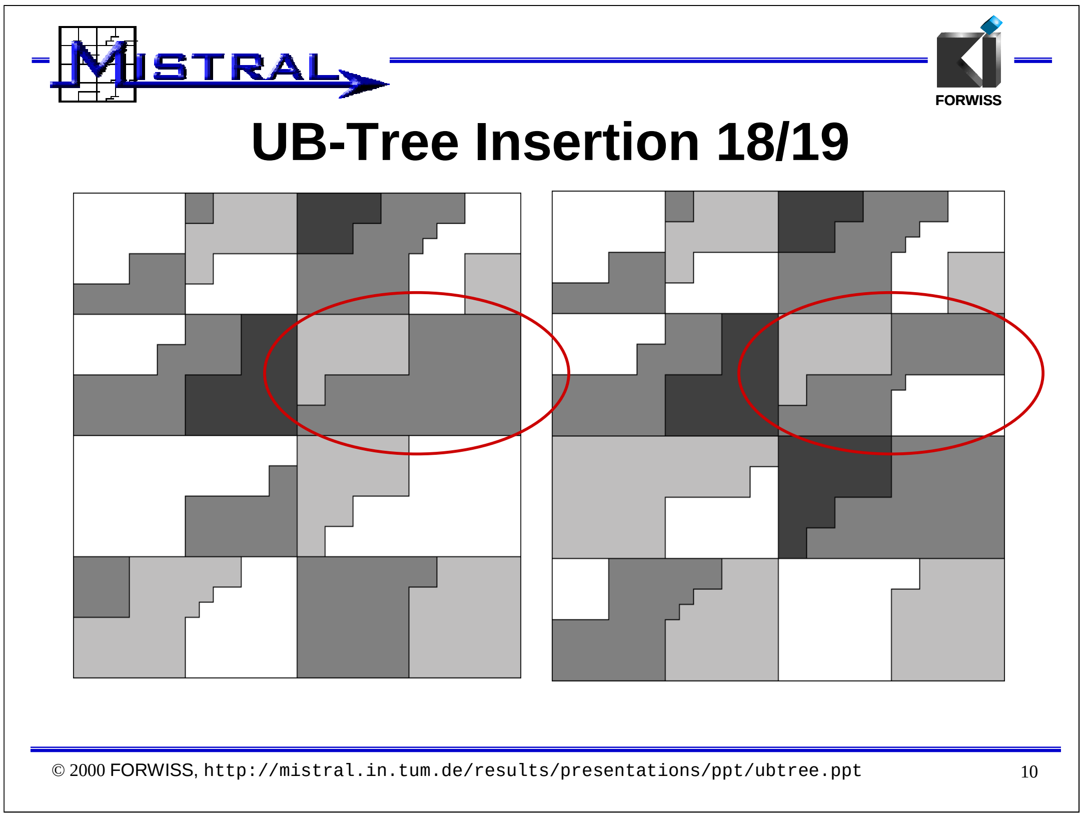
This animation shows how data is inserted into a two-dimensional UB-Tree assuming a page (Z-region) capacity of two tuples (or points). We start with an empty UB-Tree consisting of a single Z-region corresponding to one disk page (the root node of the UB-Tree). Now points are inserted into the universe (indicated by red circles). As soon as the third point is inserted, the Z-region covering the the entire universe must be split into two Z-regions. This is done by choosing a separator Z address on that page which ensures that 50% of the tuples stored already on that page will have a lower value (in Z-order) than the separator. Choosing this separator for the split ensures a page utilization of 50%. To optimize the geometric shape of Z-regions, an additional heuristics exploiting the remaining freedom of choice for the separator may be used to avoid fringes (however, this optimization is not used in the animation above). In addition, the geometric shape of the region may be improved (be more rectangular) by lowering the page utilization and thereby offering a greater freedom of choice for the separator selection. However, in practice one can remain with the 50% choice.

After the (in this case horizontal) split, the UB-Tree consists of two Z-regions. Inserting further points causes further splits each time the page capacity of a Z-region is exceeded. This animation continues until a UB-Tree consisting of 4 Z-regions storing 8 tuples has been created.

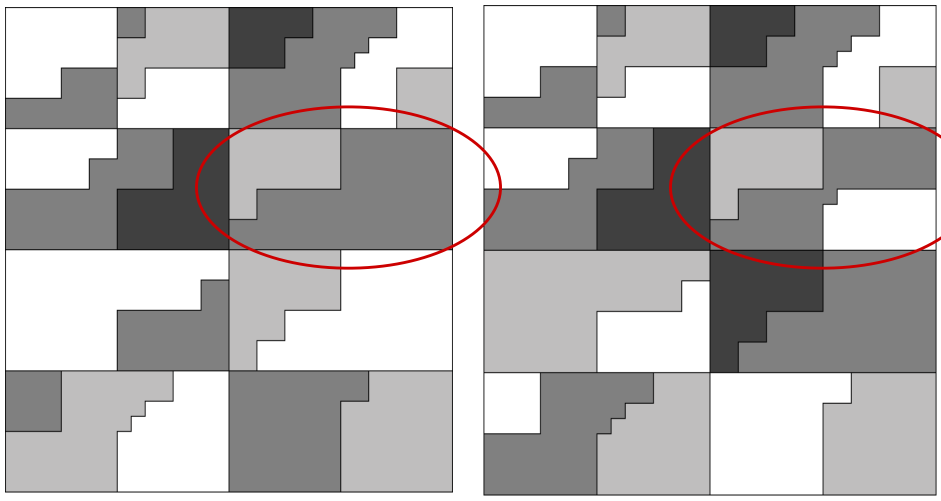
In the enclosed file

`http://mistral.in.tum.de/results/presentations/ppt/insert.ppt`

one can find another animation of the UB-Tree insertion, using our standard view with a screen split into three sections CODE, Z-SPACE and GEOMETRIC SPACE (in this presentation the range query animation uses that layout) which allows to see the algorithm running view the code, the events happening in geometric space as well as the events happening in linear Z-space as stored in the B-Tree of the UB-Tree.

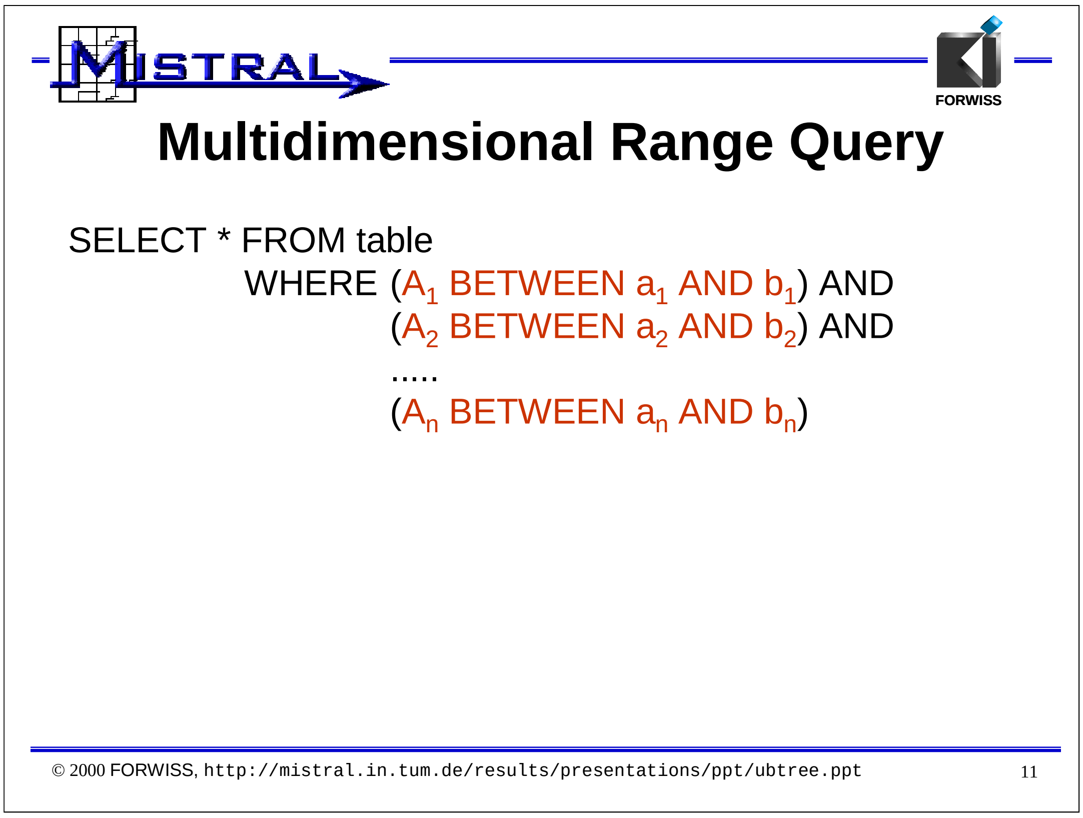


UB-Tree Insertion 18/19



10

This slide shows that insertion into UB-Trees is a local operation and thus allows for efficient incremental updates. We have inserted further points into the UB-Tree which induced further splits into 18 Z-regions (left UB-Tree visualization). The UB-Tree on the right side was created by inserting further points into the last Z-region (in Z-order) of the second quadrant (in the UB-Tree on the left). It is important to note that only this single Z-region is split, resulting in one update and one write of B-Tree leaf pages (plus possible further splits on the upper B-Tree levels). This is a major difference to other multidimensional access methods like R-Trees or Grid-Files.



Multidimensional Range Query

SELECT * FROM table

WHERE (A_1 BETWEEN a_1 AND b_1) AND

(A_2 BETWEEN a_2 AND b_2) AND

.....

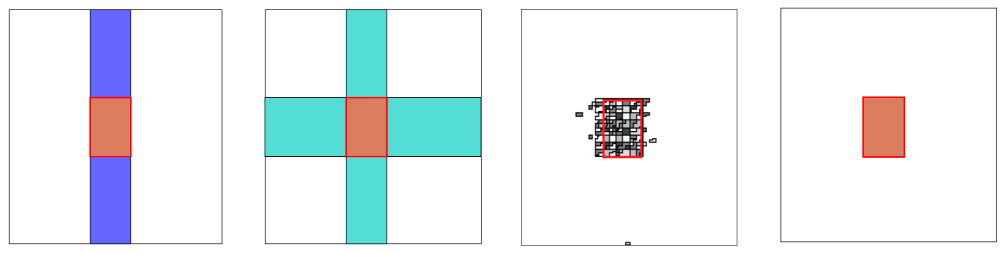
(A_n BETWEEN a_n AND b_n)

The goal of a multidimensional access methods is to efficiently answer multidimensional range queries which are created by the SQL query template above.

MISTRAL

FORWISS

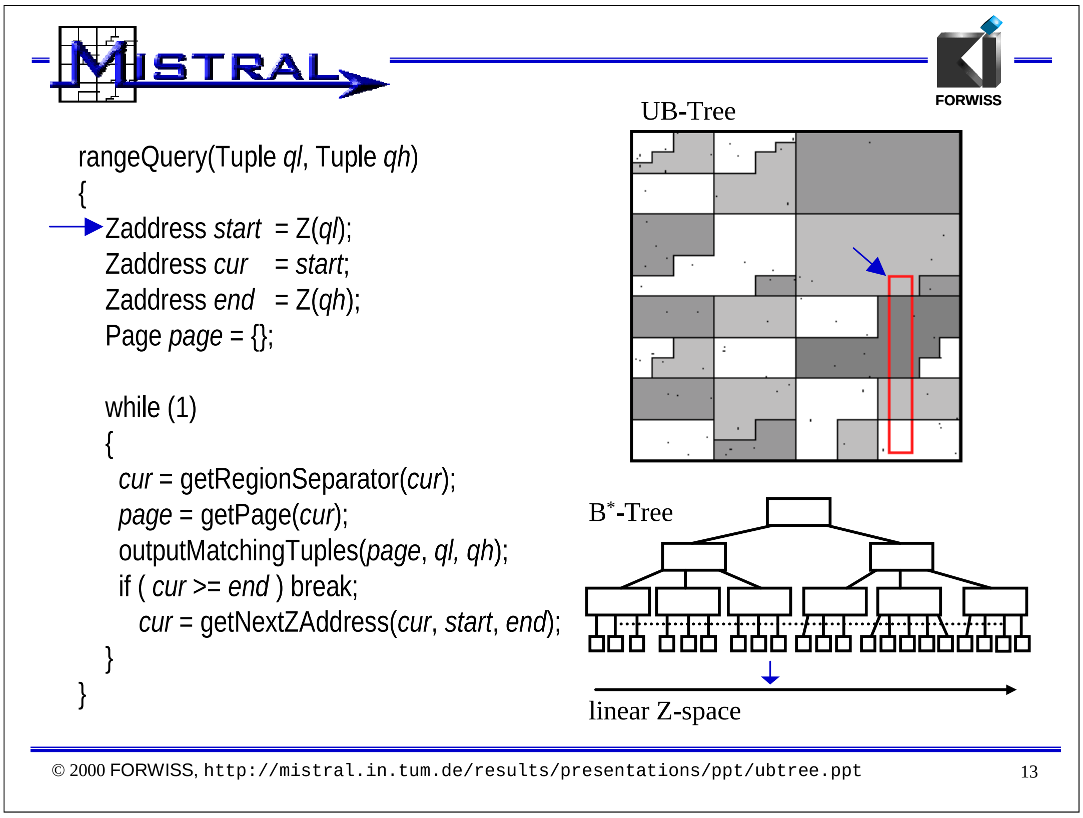
Theoretical Comparison of the Rangequery Performance



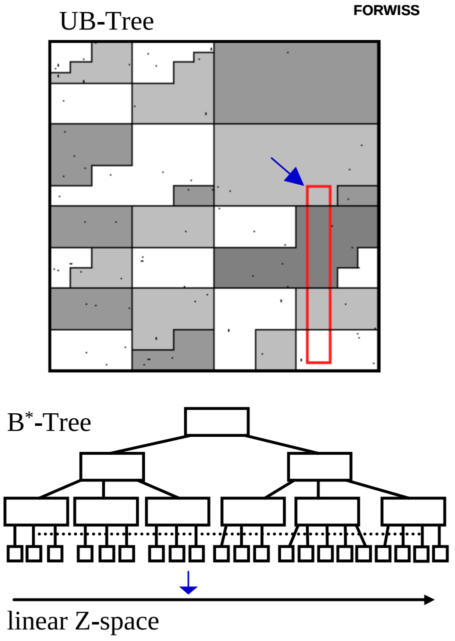
composite key clustering B-Tree	multiple B-Trees, bitmap indexes	multidimensional index	ideal case
s_1*P	$s_1*I_1+s_2*I_2+s_1*s_2*T$	$s_1^{\uparrow}*s_2^{\uparrow}*P$	s_1*s_2*P

12

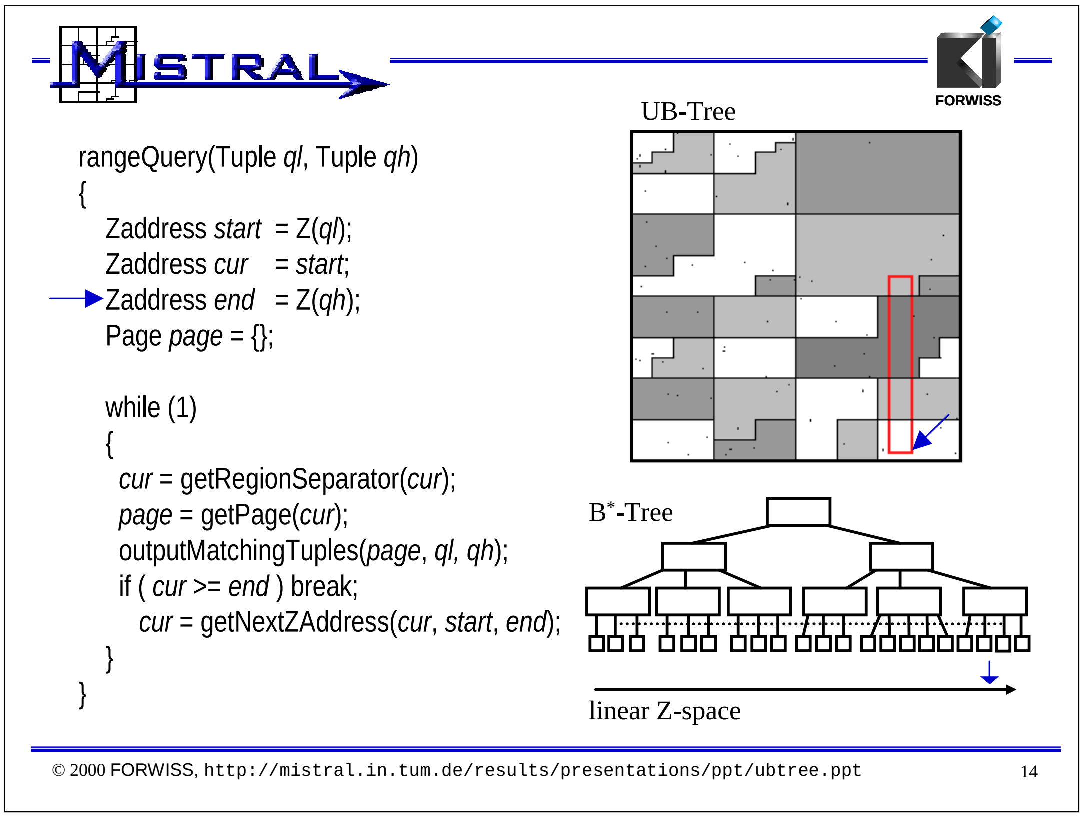
This slide illustrates how range queries are processed with access methods that are standard in today‘s relational DBMS. For simplification of our illustration we assume uniformly distributed data as well as independence of the dimensions. We assume a table consisting of P disk pages and a query box with the selectivities s1 and s2. In the ideal case we thus have to retrieve s1*s2*P disk pages to answer the query. With a composite key B-Tree, however, only the leading dimension of the composite key can be utilized, resulting in reading s1*P disk pages in the blue stripe. The result set is then determined by post filtering the tuples in main memory after retrieval. With bitmap indexes or multiple B-Trees, index intersection results in reading row ids or bitmaps with sizes s1*i1 and s2*i2 for index sizes of i1 respectively i2 pages. After this intersection, the result set tuples are retrieved by random access, resulting in s1*s2*T page reads, if T tuples are stored in the table. Note the difference between T (the number of tuples in the table) and P (the number of pages in the table). With an average of 30 tuples (empirical value from our project partners) per page bitmap indexes or multiple B-Trees are immediately more than 30 times worse than the ideal case. In contrast to that, a multidimensional index clusters the data more symmetrically with respect to all dimensions. Since this clustering or partitioning is discrete, there is always an overhead. However, with large database sizes the overhead gets smaller. In general this means that multidimensional indexes approximate the ideal case with some kind of ceiling function for each selectivity.



```
rangeQuery(Tuple ql, Tuple qh)
{
→ Zaddress start  = Z(ql);
  Zaddress cur    = start;
  Zaddress end    = Z(qh);
  Page page = {};

  while (1)
  {
    cur = getRegionSeparator(cur);
    page = getPage(cur);
    outputMatchingTuples(page, ql, qh);
    if ( cur >= end ) break;
      cur = getNextZAddress(cur, start, end);
  }
}
```



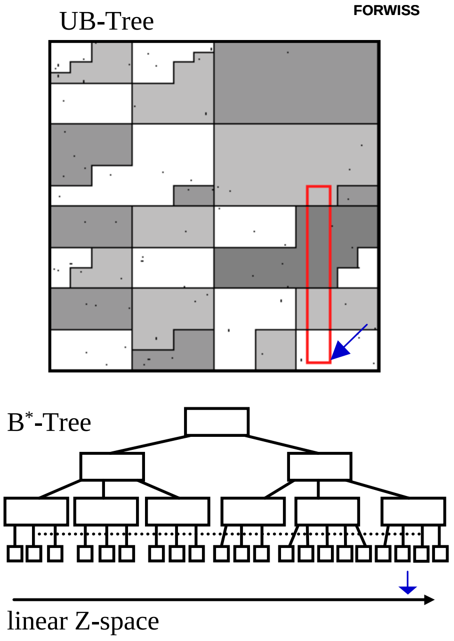
This slide and the following slides explain the UB-Tree range query algorithm. The screen is divided into 4 sections: The left part shows the code, the right part shows the geometrical interpretation of the UB-Tree (UB-Tree), the linear B*-Tree as well as the linear Z-space. Please note that the UB-Tree and the B*-Tree on the linear Z-space are merely two different interpretations or visualizations of the same data set. In the algorithms we can arbitrarily switch between both representations. In the following we show how the read query box defined by the tuples ql and qh with (ql, qh) is processed by the range query algorithm.



MISTRAL

FORWISS

```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
→  Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
        cur = getNextZAddress(cur, start, end);
   }
}
```

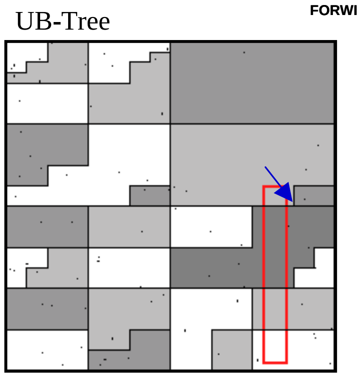


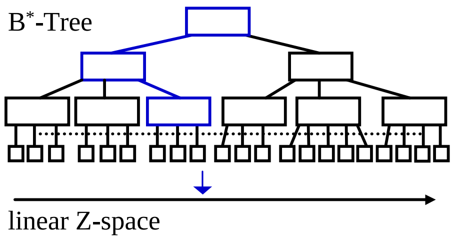
First the algorithm calculates the start and the end Z-values of the query box coordinates ql and qh.

MISTRAL

FORWISS

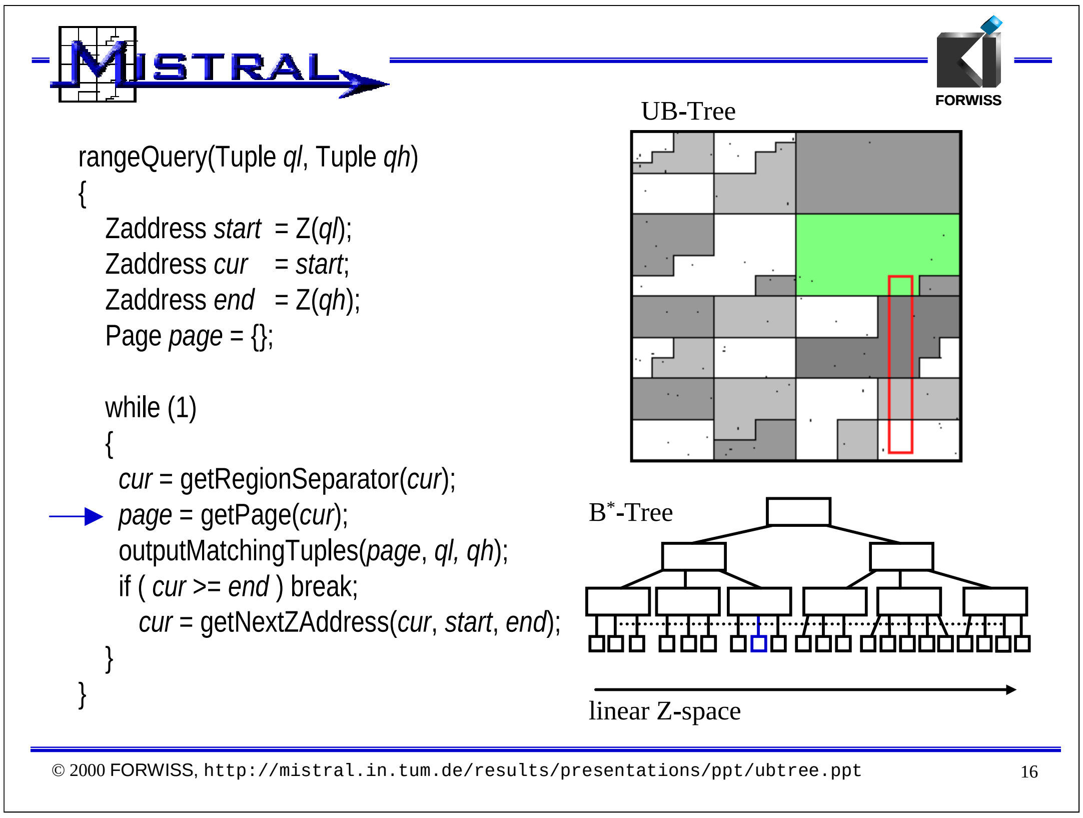
```
rangeQuery(Tuple ql, Tuple qh)
{
    Zaddress start  = Z(ql);
    Zaddress cur    = start;
    Zaddress end    = Z(qh);
    Page page = {};

    while (1)
    {
      cur = getRegionSeparator(cur);
      page = getPage(cur);
      outputMatchingTuples(page, ql, qh);
      if ( cur >= end ) break;
         cur = getNextZAddress(cur, start, end);
    }
}
```





15

First the region separator of the Z-region where the start point of the query box is located is determined by a single B-Tree point search (in SQL: SELECT min(Z) where Z>cur) as illustrated by the blue path through the B-Tree.

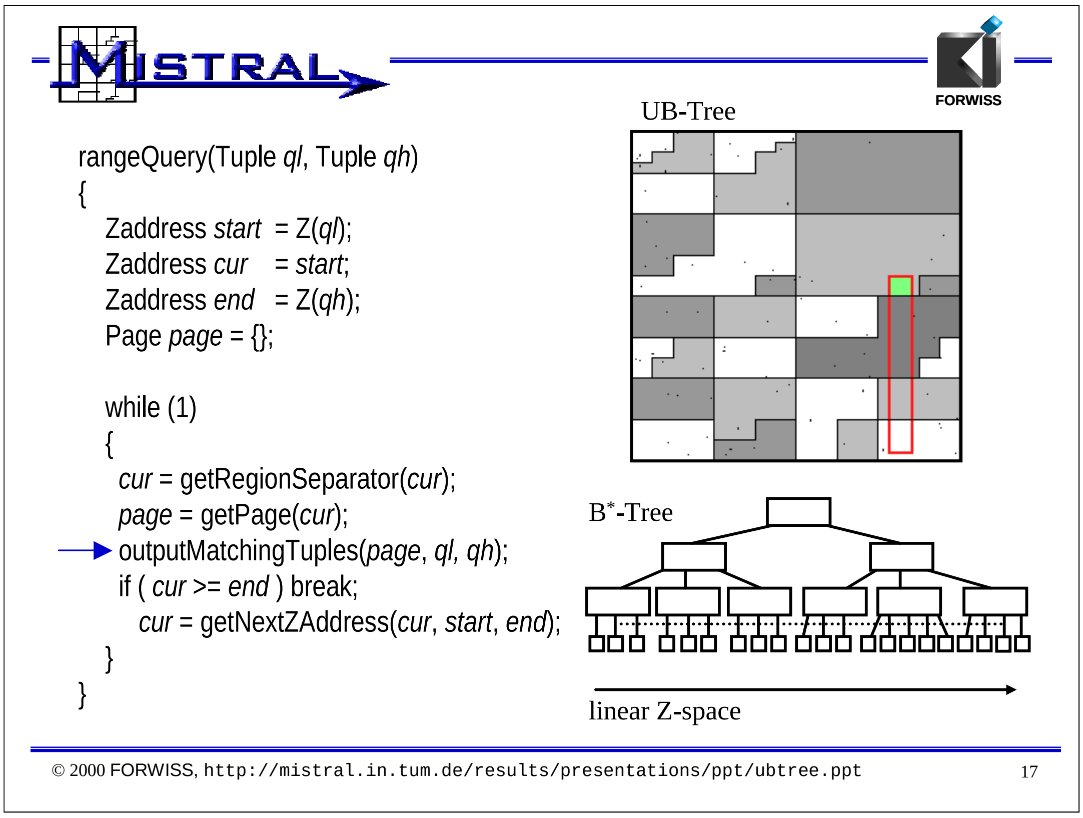


```
rangeQuery(Tuple ql, Tuple qh)
{
    Zaddress start  = Z(ql);
    Zaddress cur    = start;
    Zaddress end    = Z(qh);
    Page page = {};

    while (1)
    {
      cur = getRegionSeparator(cur);
  ──► page = getPage(cur);
      outputMatchingTuples(page, ql, qh);
      if ( cur >= end ) break;
         cur = getNextZAddress(cur, start, end);
    }
}
```

MISTRAL
FORWISS
UB-Tree
B*-Tree
linear Z-space

© 2000 FORWISS, `http://mistral.in.tum.de/results/presentations/ppt/ubtree.ppt` 16

Then the leaf page corresponding to that Z-region is retrieved as illustrated by the blue leaf page in the B-Tree as well as the green colored space in the UB-Tree.

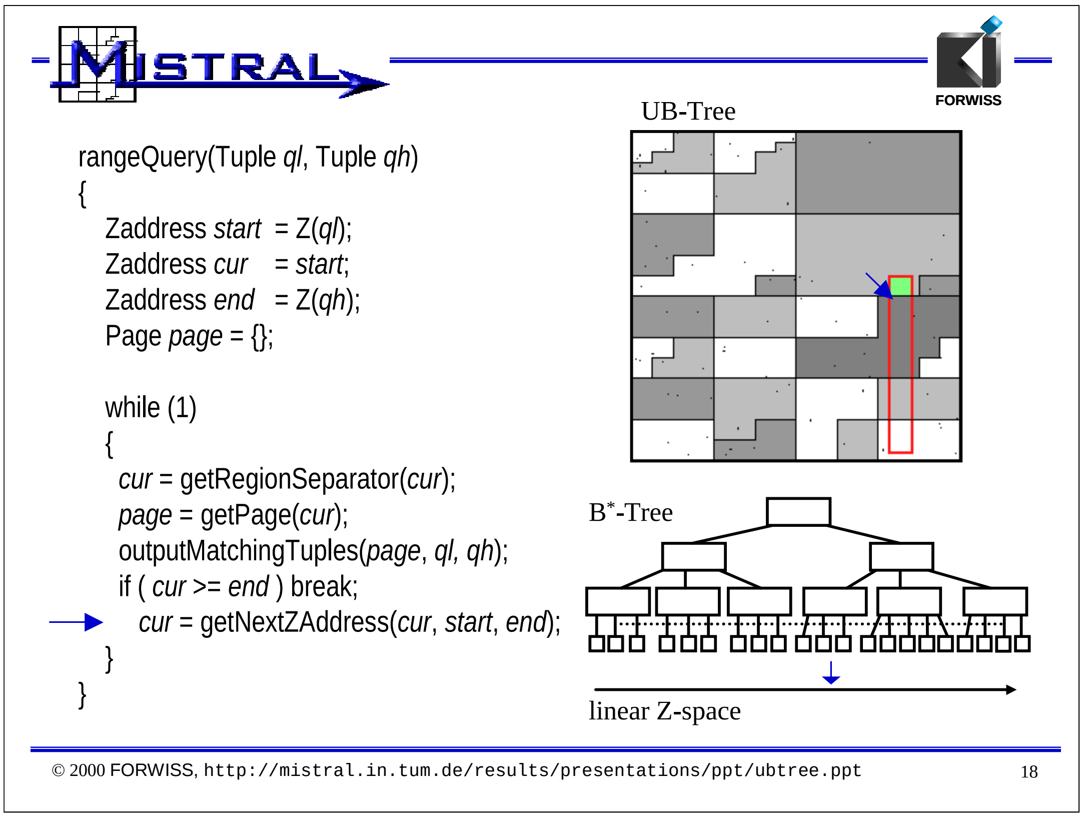


```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
──▶  outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
        cur = getNextZAddress(cur, start, end);
   }
}
```

UB-Tree
B*-Tree
linear Z-space

17

Then all tuples of that page that are inside the query box are returned.



```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
-->    cur = getNextZAddress(cur, start, end);
   }
}
```

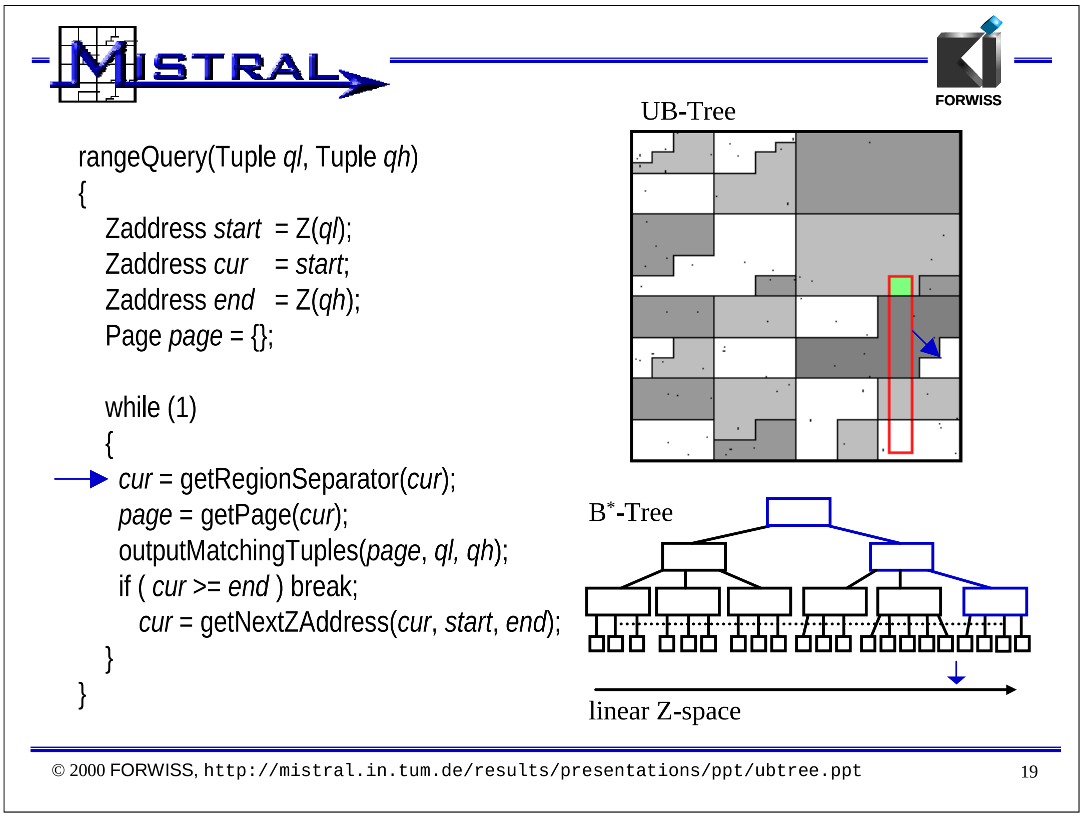
© 2000 FORWISS, `http://mistral.in.tum.de/results/presentations/ppt/ubtree.ppt` 18

Now the next Z-address intersecting the query box is computed, i.e.,

cur = min { Z(x) where x in [[ql, qh]] and Z(x) > cur)

This is achieved by an algorithm that only requires O(n) bit operations (copy and compare) where n is the number of bits (i.e., length) of the Z-address cur.

The details of that algorithm can be found in [Mar99] and [RMF+00].

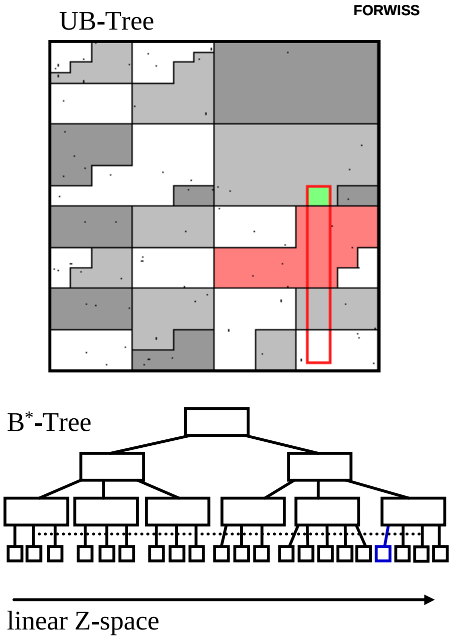


```
rangeQuery(Tuple ql, Tuple qh)
{
    Zaddress start  = Z(ql);
    Zaddress cur    = start;
    Zaddress end    = Z(qh);
    Page page = {};

    while (1)
    {
→     cur = getRegionSeparator(cur);
      page = getPage(cur);
      outputMatchingTuples(page, ql, qh);
      if ( cur >= end ) break;
         cur = getNextZAddress(cur, start, end);
    }
}
```

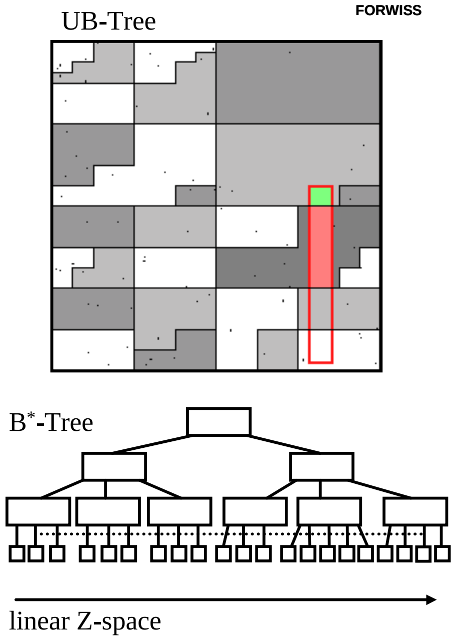
UB-Tree
B*-Tree
linear Z-space

After that the algorithm proceeds in the same way as described before until the entire query box has been processed, i.e., cur > end.

```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
→    page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
        cur = getNextZAddress(cur, start, end);
   }
}
```



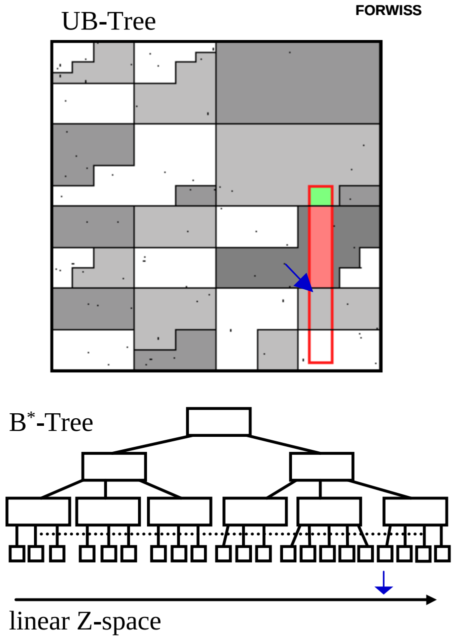
```
rangeQuery(Tuple ql, Tuple qh)
{
    Zaddress start  = Z(ql);
    Zaddress cur    = start;
    Zaddress end    = Z(qh);
    Page page = {};

    while (1)
    {
      cur = getRegionSeparator(cur);
      page = getPage(cur);
      outputMatchingTuples(page, ql, qh);
      if ( cur >= end ) break;
         cur = getNextZAddress(cur, start, end);
    }
}
```

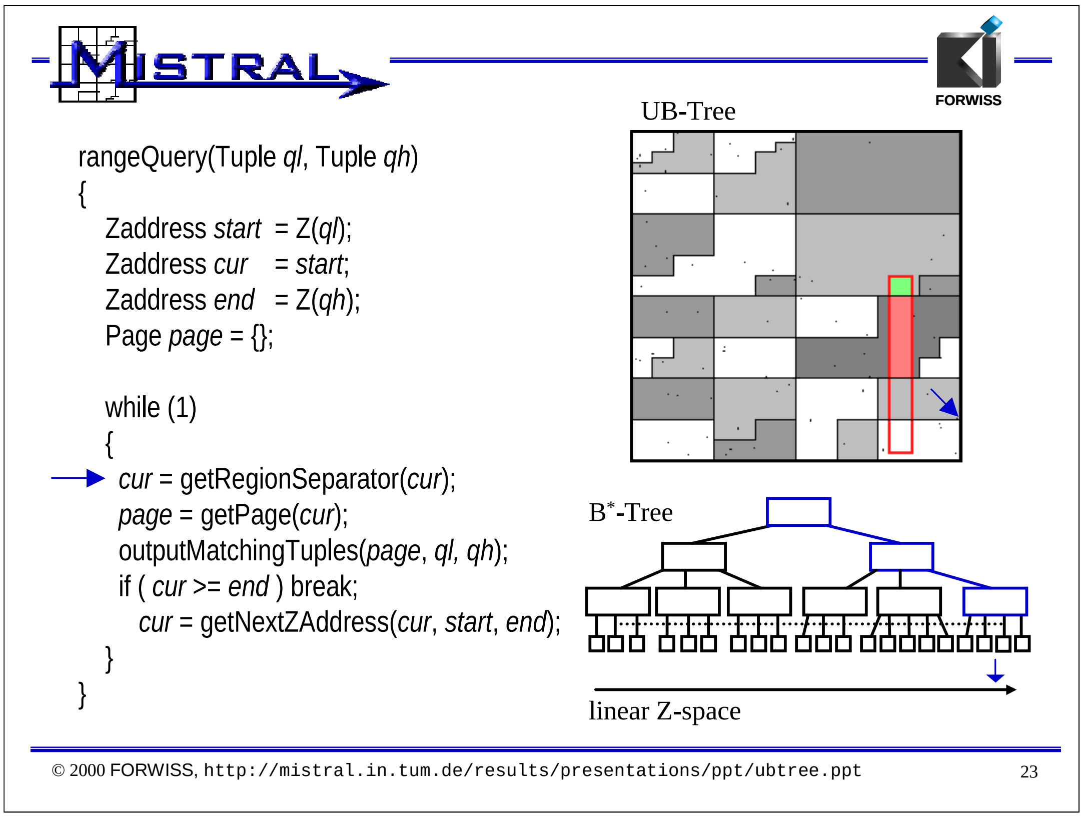


MISTRAL

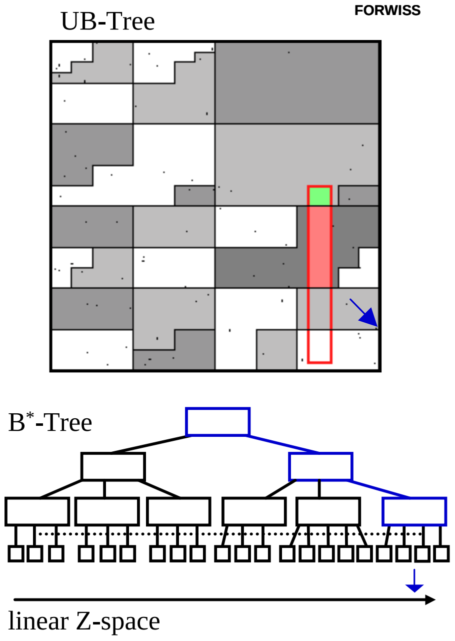
FORWISS

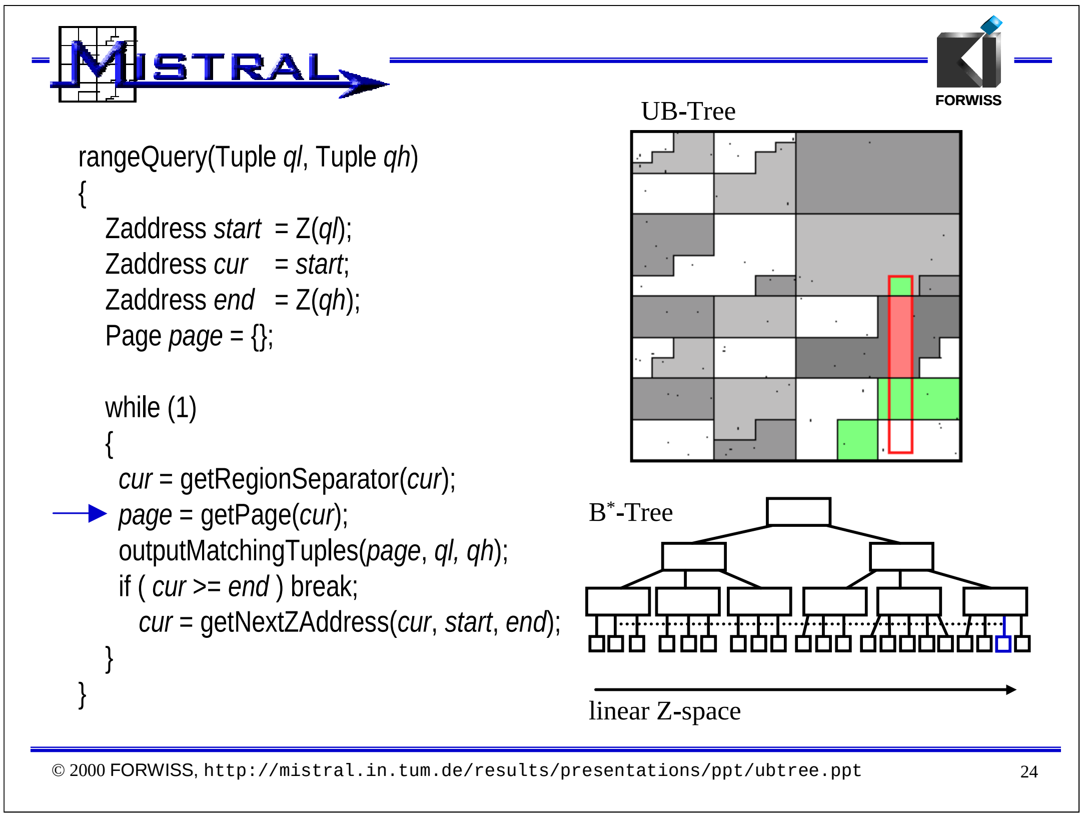
```
    rangeQuery(Tuple ql, Tuple qh)
    {
       Zaddress start  = Z(ql);
       Zaddress cur    = start;
       Zaddress end    = Z(qh);
       Page page = {};

       while (1)
       {
         cur = getRegionSeparator(cur);
         page = getPage(cur);
         outputMatchingTuples(page, ql, qh);
         if ( cur >= end ) break;
→           cur = getNextZAddress(cur, start, end);
       }
    }
```



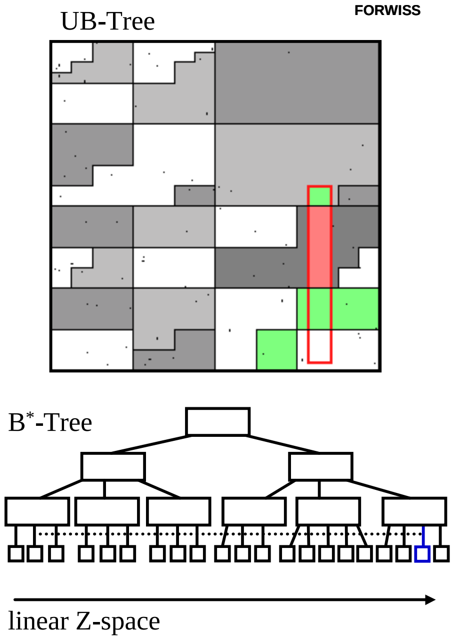


```
rangeQuery(Tuple ql, Tuple qh)
{
    Zaddress start  = Z(ql);
    Zaddress cur    = start;
    Zaddress end    = Z(qh);
    Page page = {};

    while (1)
    {
→     cur = getRegionSeparator(cur);
      page = getPage(cur);
      outputMatchingTuples(page, ql, qh);
      if ( cur >= end ) break;
         cur = getNextZAddress(cur, start, end);
    }
}
```

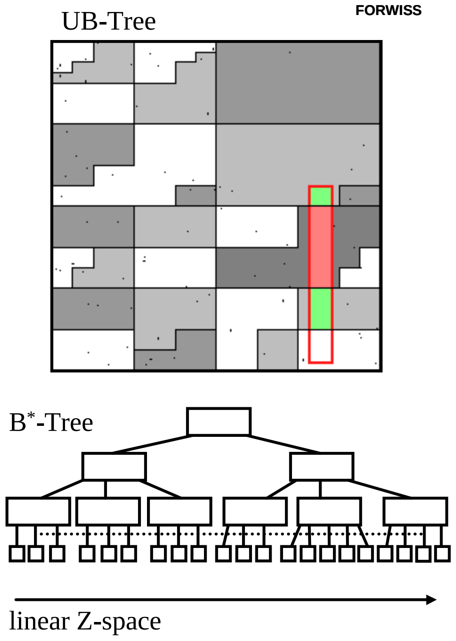




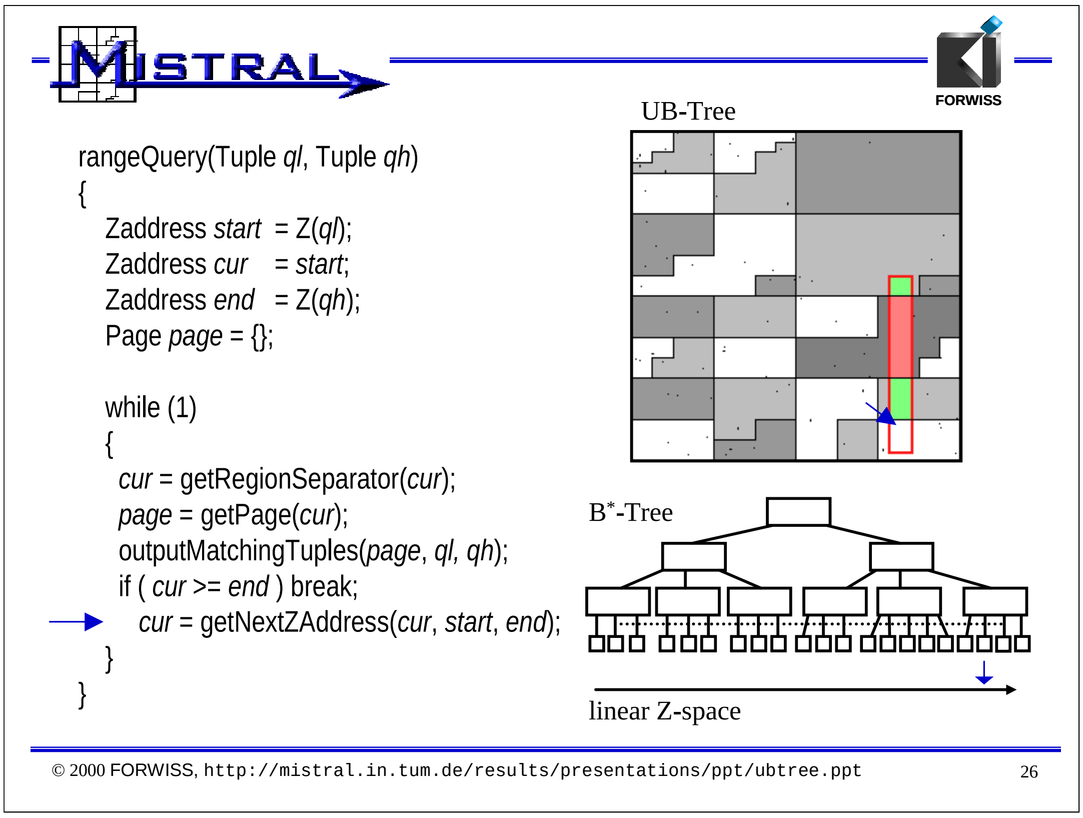
```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
        cur = getNextZAddress(cur, start, end);
   }
}
```



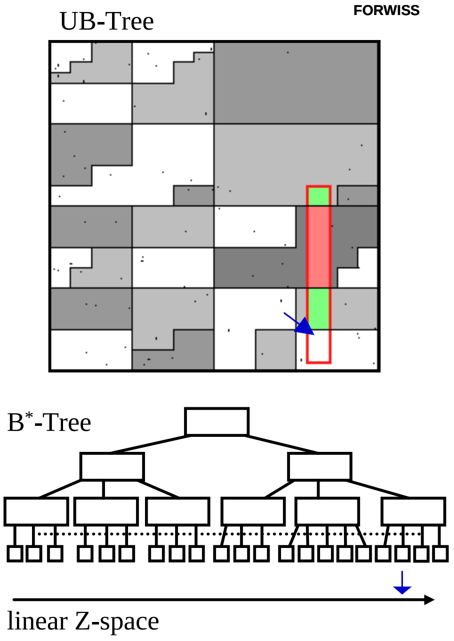
```
rangeQuery(Tuple ql, Tuple qh)
{
    Zaddress start  = Z(ql);
    Zaddress cur    = start;
    Zaddress end    = Z(qh);
    Page page = {};

    while (1)
    {
      cur = getRegionSeparator(cur);
      page = getPage(cur);
  →   outputMatchingTuples(page, ql, qh);
      if ( cur >= end ) break;
        cur = getNextZAddress(cur, start, end);
    }
}
```



25



```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
→      cur = getNextZAddress(cur, start, end);
   }
}
```

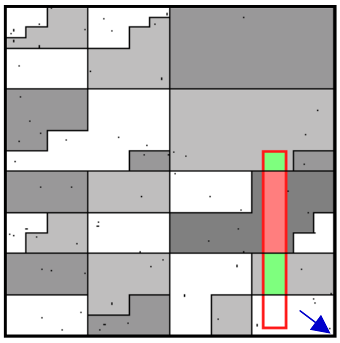


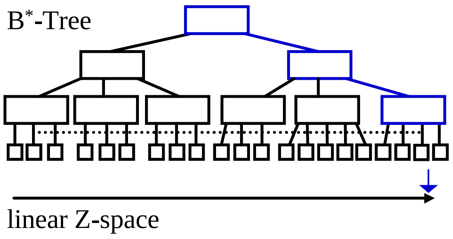
MISTRAL

FORWISS

```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
        cur = getNextZAddress(cur, start, end);
   }
}
```

UB-Tree



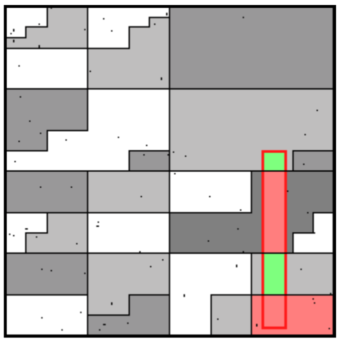


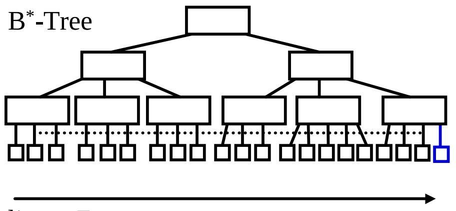
MISTRAL

FORWISS

```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
        cur = getNextZAddress(cur, start, end);
   }
}
```

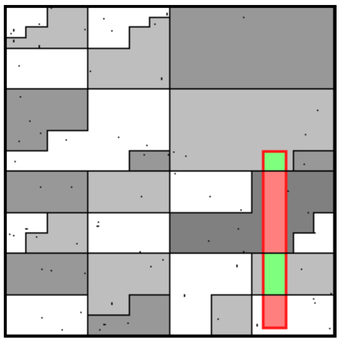
UB-Tree

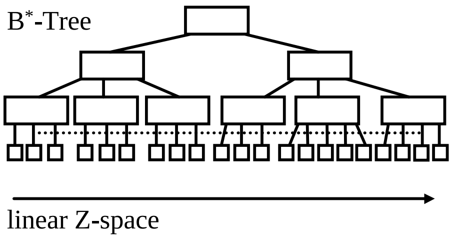


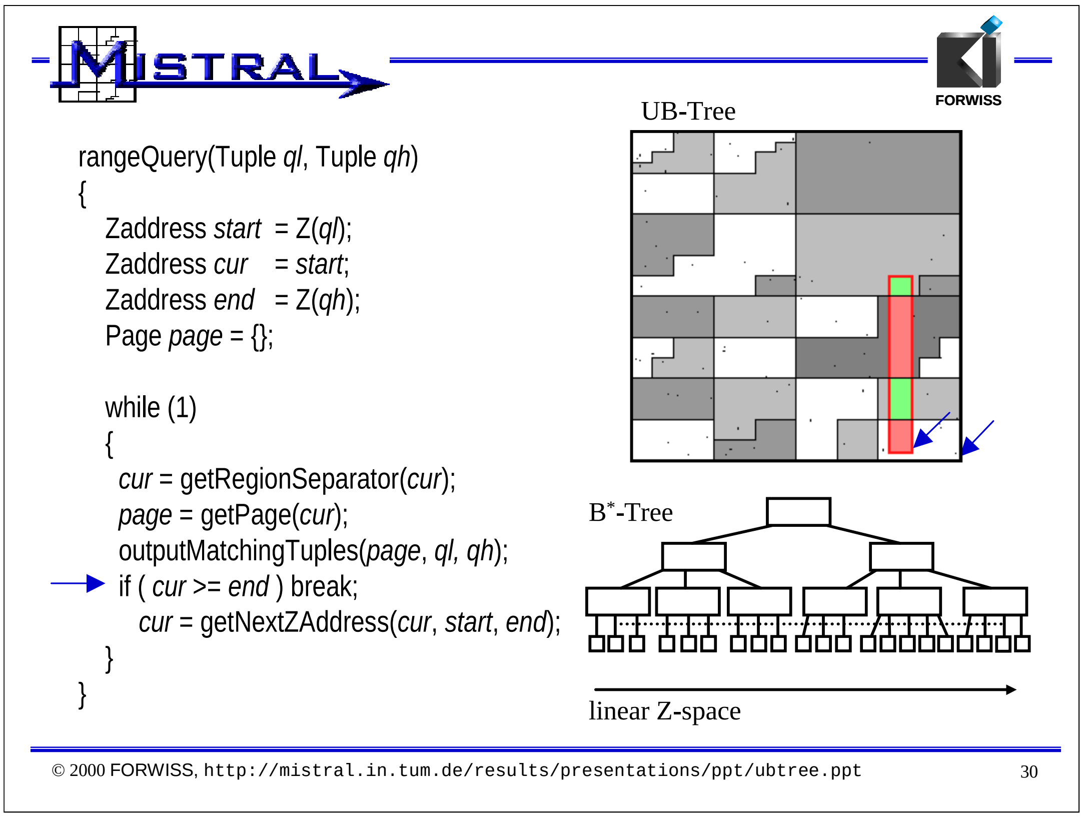


```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
→    outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
       cur = getNextZAddress(cur, start, end);
   }
}
```

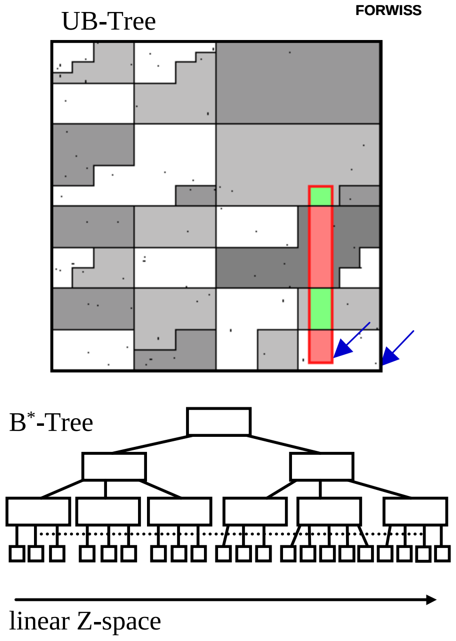
UB-Tree



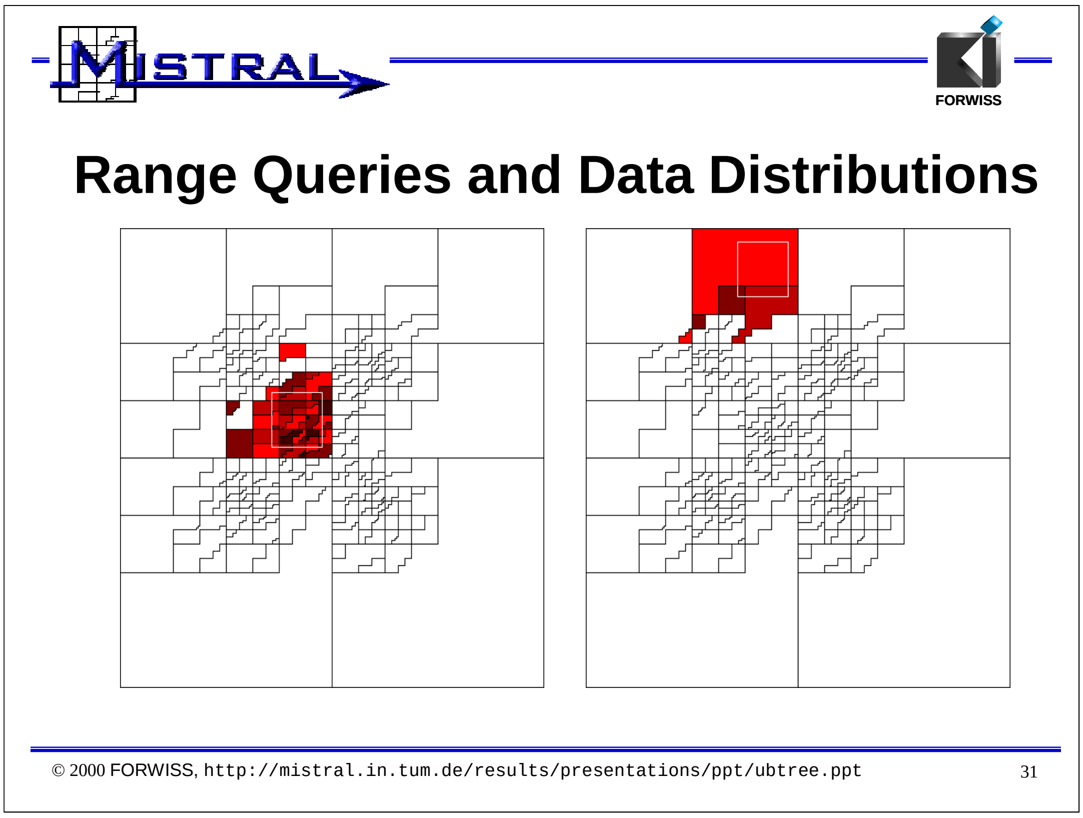




```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
→    if ( cur >= end ) break;
       cur = getNextZAddress(cur, start, end);
   }
}
```

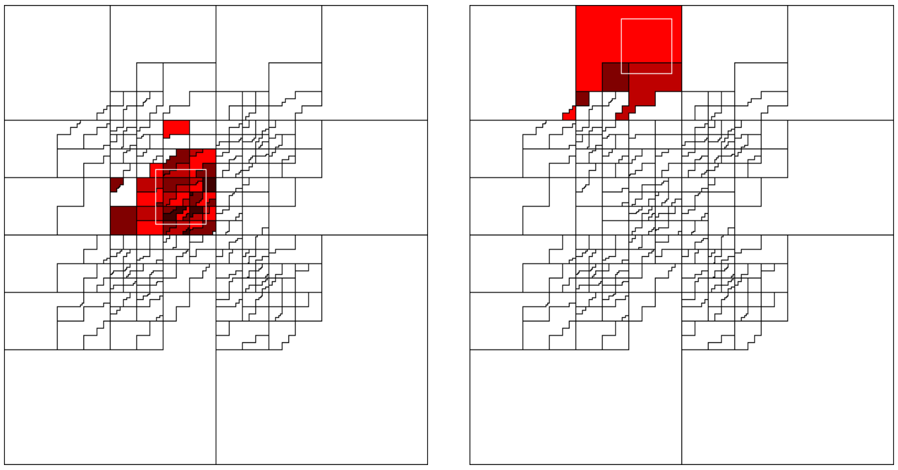


Now the end condition is satisfied, since the largest address of the Z-region is larger than the end address of the query box. Thus, the algorithm can terminate now.

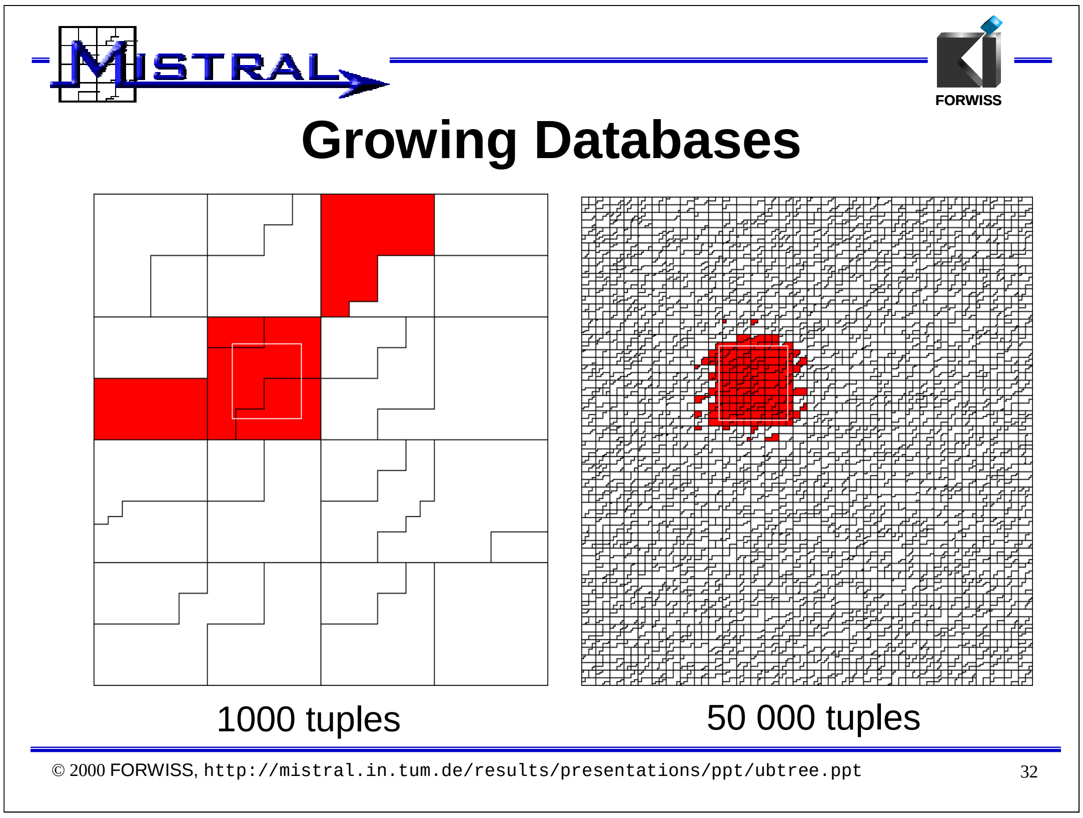


FORWISS

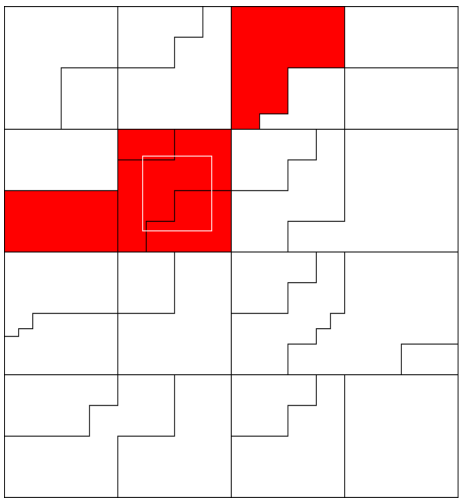
Range Queries and Data Distributions



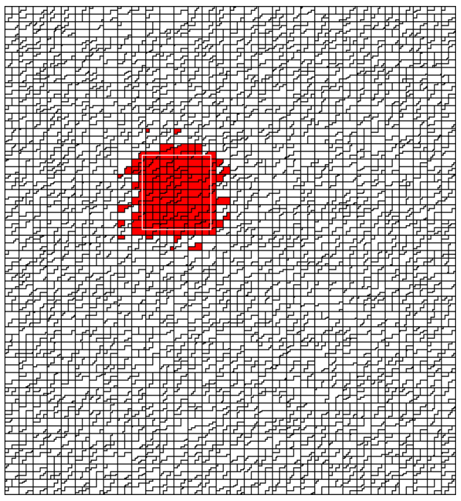
This slide shows range queries in sparsely and densely populated parts of the universe. The data of this UB-Tree consists of 5 clusters (please note that small Z-regions denote parts of the space that are densely populated since each Z-region stores about the same number of tuples). Whereas a query in a densely populated part of the space (left side) retrieves a lot of Z-regions, a query in the sparsely populated part (right side) retrieves only 3 Z-regions. This means that the number of pages retrieved is correlated with the results set size, resulting in a very desirable response time behavior.



Growing Databases



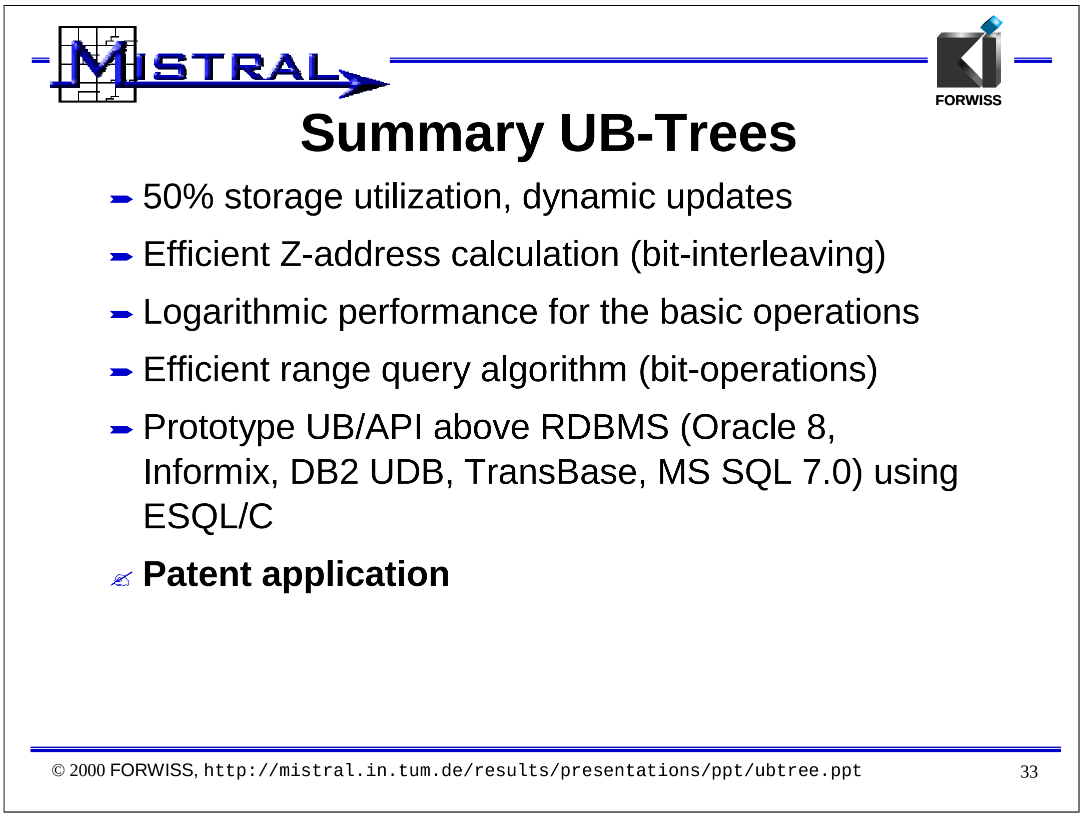
1000 tuples



50 000 tuples

© 2000 FORWISS, `http://mistral.in.tum.de/results/presentations/ppt/ubtree.ppt` 32

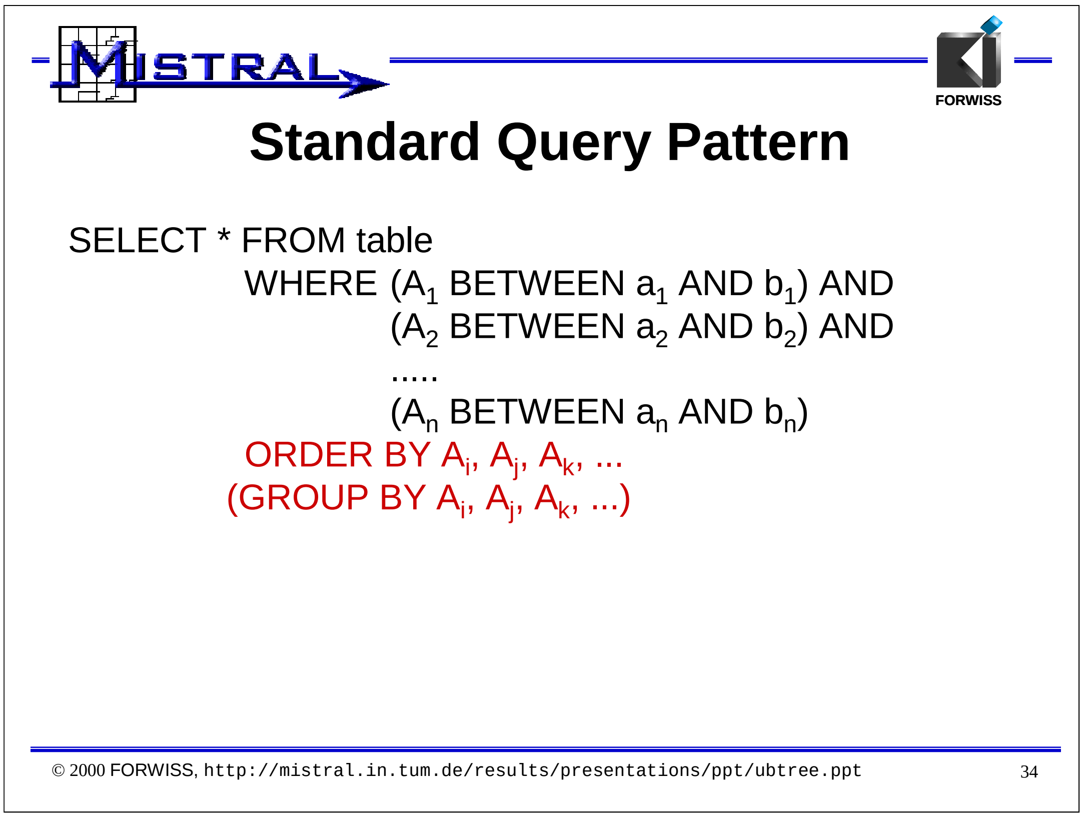
This slide shows that with larger database sizes the Z-region partitioning gets finer and query boxes are better approximated by the partitioning. This slide also shows that the UB-Tree partitioning will very well answer a range query if the query box is larger than the average Z-region size. Of course, if the query box is smaller than a Z-region, only one or very few Z-regions must be retrieved.



Summary UB-Trees

- 50% storage utilization, dynamic updates
- Efficient Z-address calculation (bit-interleaving)
- Logarithmic performance for the basic operations
- Efficient range query algorithm (bit-operations)
- Prototype UB/API above RDBMS (Oracle 8, Informix, DB2 UDB, TransBase, MS SQL 7.0) using ESQL/C
- ✍ **Patent application**

This slide summarizes the achievements of the MISTRAL project with respect to the UB-Tree prototype implementation.

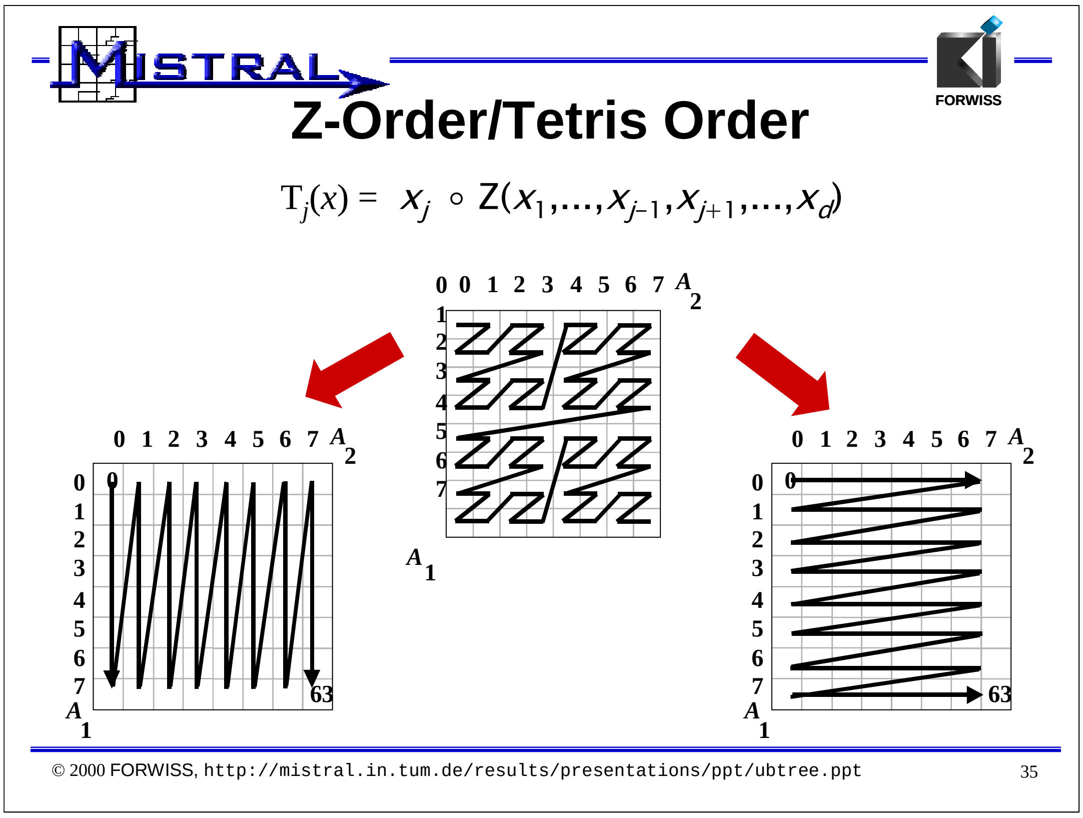


FORWISS

Standard Query Pattern

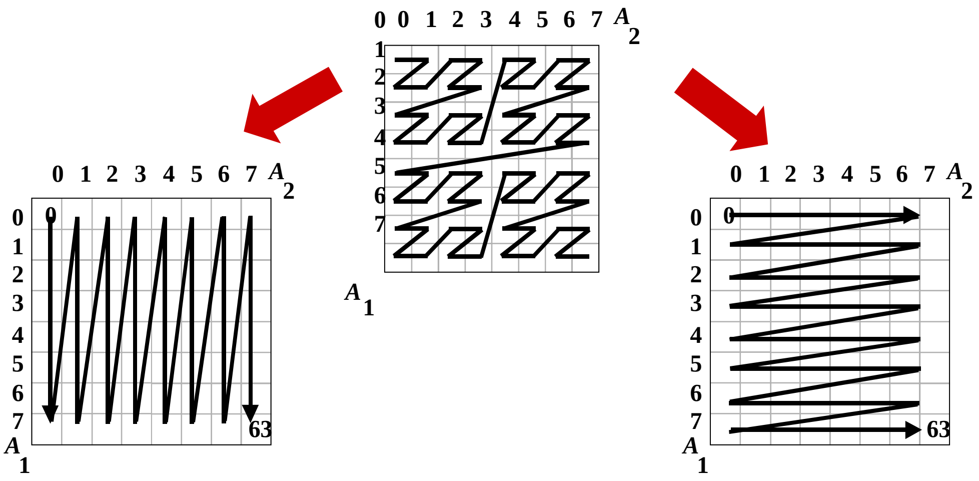
SELECT * FROM table
WHERE (A_1 BETWEEN a_1 AND b_1) AND
(A_2 BETWEEN a_2 AND b_2) AND
.....
(A_n BETWEEN a_n AND b_n)
ORDER BY A_i, A_j, A_k, ...
(GROUP BY A_i, A_j, A_k, ...)

In addition to range queries, sorting is a very important and frequent operation in relational databases. Sorting is not only used for ordering, but also provides a basis for efficient algorithms for duplicate elimination or join operations as well as for group by operations.



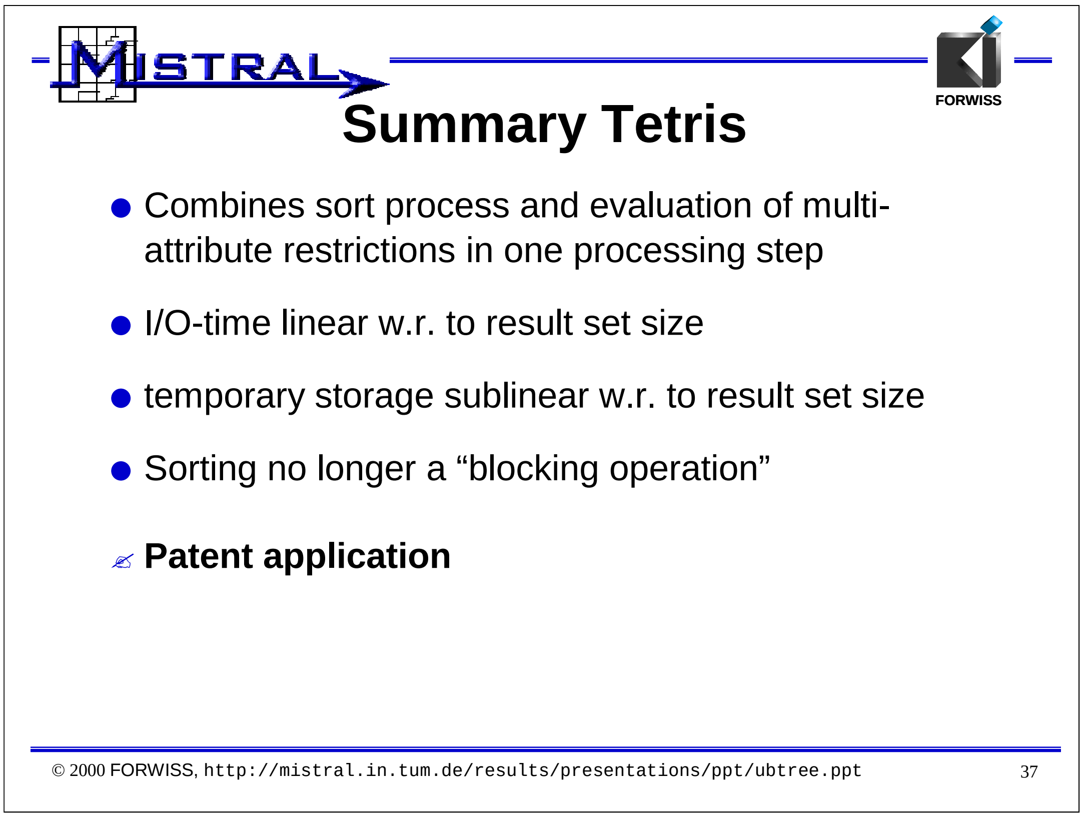
Z-Order/Tetris Order

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



© 2000 FORWISS, `http://mistral.in.tum.de/results/presentations/ppt/ubtree.ppt` 35

Sorting a Z-ordered space means to introduce a Tetris order, an ordering that extracts a single attribute out of a Z-ordered space. The Z-regions are read in this Tetris order as opposed to the Z-order that the range query uses to retrieve Z-regions. Please confer to [MZB99] for detailed information about Tetris order and the Tetris algorithm.

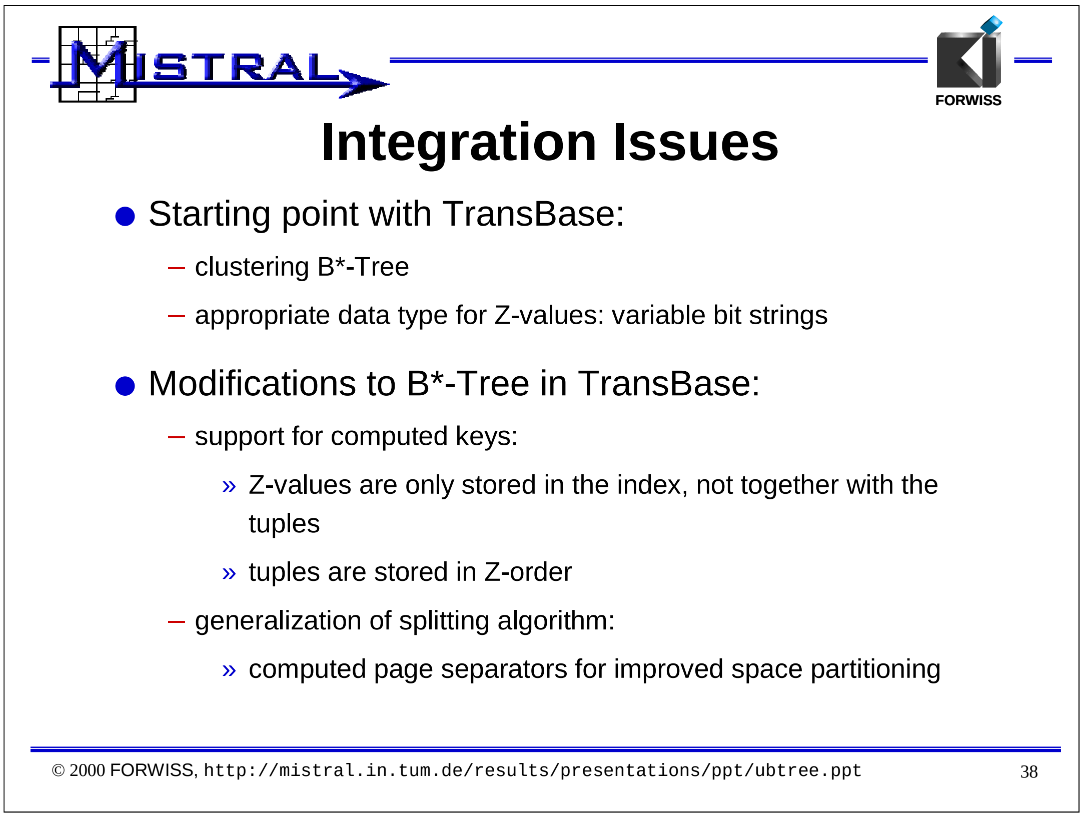


Summary Tetris

- Combines sort process and evaluation of multi-attribute restrictions in one processing step
- I/O-time linear w.r. to result set size
- temporary storage sublinear w.r. to result set size
- Sorting no longer a “blocking operation”

✍ **Patent application**

This slide summarizes the main results of the Tetris algorithm.



Integration Issues

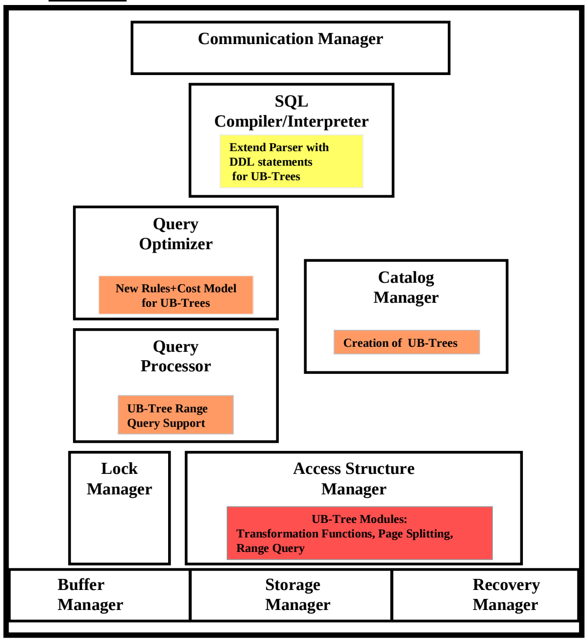
- Starting point with TransBase:
  - clustering B*-Tree
  - appropriate data type for Z-values: variable bit strings
- Modifications to B*-Tree in TransBase:
  - support for computed keys:
    - Z-values are only stored in the index, not together with the tuples
    - tuples are stored in Z-order
  - generalization of splitting algorithm:
    - computed page separators for improved space partitioning

In the MDA project between FORWISS, GfK, and TransAction Software the UB-Tree was integrated seamlessly into the RDBMS TransBase. The resulting product TransBase HyperCube is shipping since Systems 1999 and was awarded the 2001 IT-Prize by EUROCASE and the European Commission.

The TransBase RDBMS already provided clustering B*-Trees and a bitstring datatype, which are a pre-requisite for a UB-Tree implementation. These implementations had to be slightly modified and enhanced in order to store the Z-addresses used by the UB-Tree.

MISTRAL

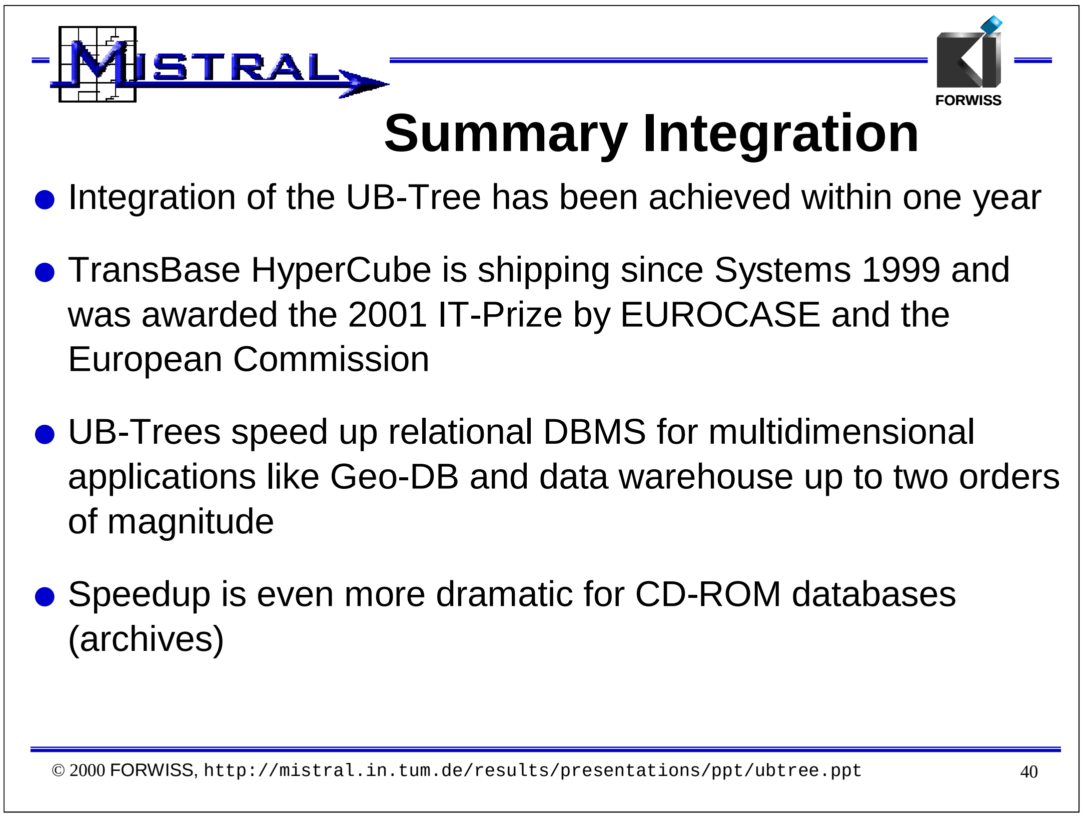
FORWISS



- Minor extensions: 🟨
- Major extensions: 🟧
- New modules: 🟥
- **NO changes for:**
  - DML
  - Multi-user support, i.e., locking, logging facilities → handled by underlying B*-Tree

39

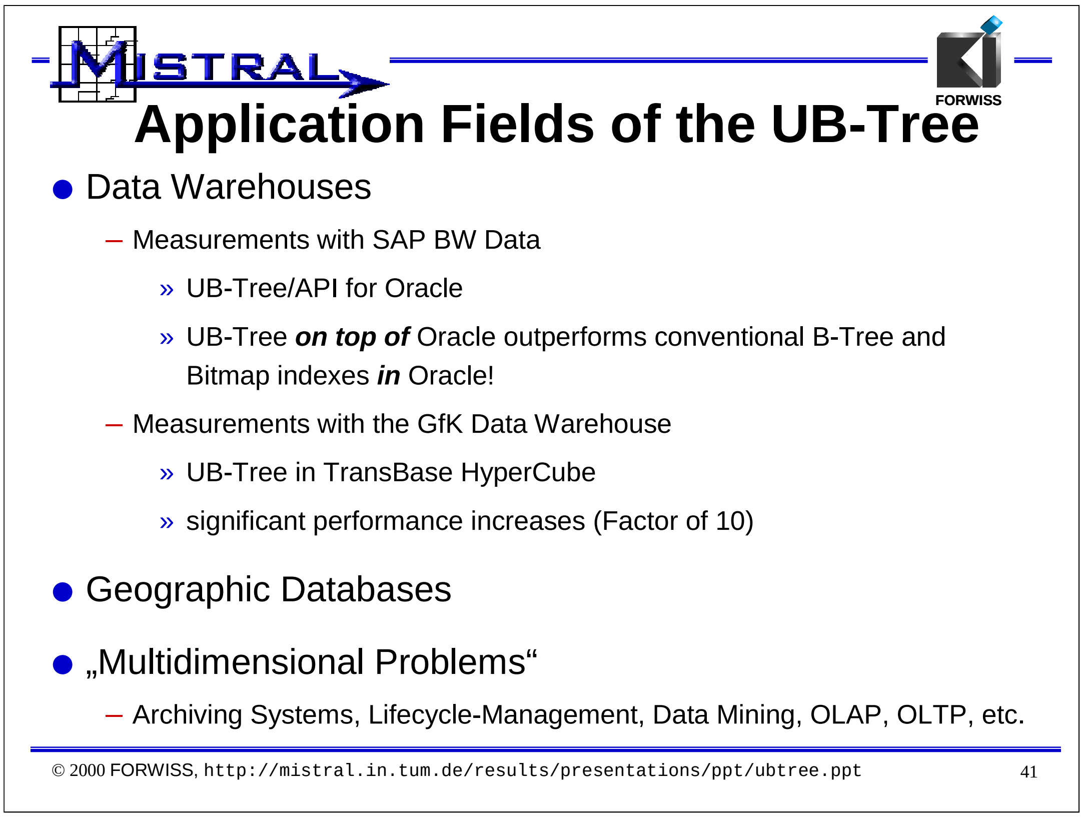
This slide shows the extensions that had to be made to TransBase in order to incorporate the UB-Tree. It is important to note that UB-Trees rely on an underlying B-Tree; thus locking, caching and recovery did not need to be modified. Further details about the kernel integration can be found in [RMF+00].



Summary Integration

- Integration of the UB-Tree has been achieved within one year
- TransBase HyperCube is shipping since Systems 1999 and was awarded the 2001 IT-Prize by EUROCASE and the European Commission
- UB-Trees speed up relational DBMS for multidimensional applications like Geo-DB and data warehouse up to two orders of magnitude
- Speedup is even more dramatic for CD-ROM databases (archives)

This slide summarizes the integration of the UB-Tree into TransBase.



Application Fields of the UB-Tree

- Data Warehouses
  - Measurements with SAP BW Data
    - » UB-Tree/API for Oracle
    - » UB-Tree ***on top of*** Oracle outperforms conventional B-Tree and Bitmap indexes ***in*** Oracle!
  - Measurements with the GfK Data Warehouse
    - » UB-Tree in TransBase HyperCube
    - » significant performance increases (Factor of 10)
- Geographic Databases
- „Multidimensional Problems“
  - Archiving Systems, Lifecycle-Management, Data Mining, OLAP, OLTP, etc.

41

The UB-Tree can be applied to any large record set that is queried and retrieved by multidimensional range queries.

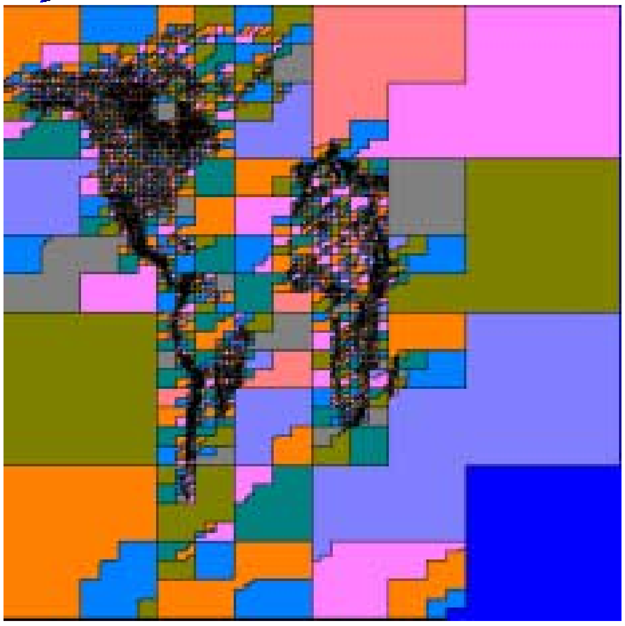
A typical application is data warehousing: Measurements with an UB-Tree API on top of Oracle compared to built-in Oracle indexes (including bitmap and IOT) showed speed-ups in favor of the UB-Tree, sometimes of more than two orders of magnitude. Similar results have been achieved with TransBase HyperCube compared to native TransBase indexes.

The product TransBase is also used for GIS database, for instance for tracking the signal quality of the cells of a mobile phone network. Further application areas for TransBase HyperCube include archiving systems, data ming, and lifecycle management. Due to the good update and multi-user characteristics, UB-Trees can also be used to organize OLTP databases.

Performance measurements details and applications of the UB-Tree and TransBase HyperCube can be found under http://mistral.in.tum.de and http://www.transaction.de

MISTRAL FORWISS

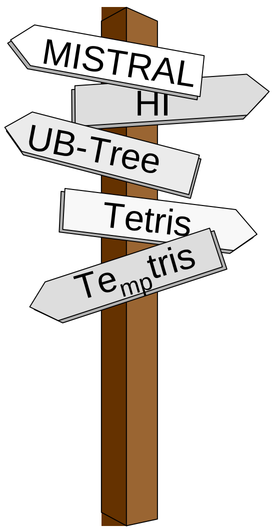
The UB-Tree



The figure above shows the Z-region partitioning for a GIS database storing point data for Africa, Europe and the Americas.

MISTRAL FORWISS

Further Information



http://mistral.in.tum.de

mistral@in.tum.de